

6.033: Computer Systems Engineering

Spring
2012

Home / News

Schedule

Submissions

Preparation for MapReduce recitation

General Information

Staff List

Recitations

TA Office Hours

- Read MapReduce.
- Skip sections 4 and 7.

This paper was published at the biennial Usenix Symposium on Operating Systems Design and Implementation (OSDI) in 2004, one of the premier conferences in computer systems. (OSDI alternates with the equally prestigious ACM Symposium on Operating Systems Principles (SOSP), at which appeared Eraser, the paper you already read in a previous recitation.)

Discussion / feedback

As you read the paper, keep the following questions in mind:

FAQ

Class Notes Errata

Excellent Writing Examples

- At first glance, the map/reduce model of computation seems limited. Did the paper persuade you that their model of computation has practical use?
- Are the authors trying to solve a technological problem (one that will be solved with faster computation), or an intrinsic problem?
- What assumptions do the authors make about how machines fail, what machines fail, and what they do when they fail? What happens to the system when a given machine fails?
- What exactly would happen if one block of one hard drive got erased during a map/reduce computation? What parts of the system would fix the error (if any), and what parts of the system would be oblivious (if any)?
- How do the authors evaluate the performance of their system? What are "Input," "Output," and "Shuffle?"
- How do "stragglers" impact performance?

2011 Home



Here are some points to keep in mind as you read:

- In the functional programming notation used in Section 2.2, the function takes the arguments shown to the left of the arrow and returns the type shown to the right of the arrow.
- After you read Section 3.1, you should be able to instantly recall the following terms: "split," "map worker," "reduce worker," "master."

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

[Top](#) // [6.033 home](#) //

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Read 3/7

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

from 6.005

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

The map function emits each word plus an associated count of occurrences (just '1' in this simple example). The reduce function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

```
map      (k1, v1)      → list (k2, v2)
reduce   (k2, list (v2)) → list (v2)
```

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle \text{URL}, 1 \rangle$. The reduce function adds together all values for the same URL and emits a $\langle \text{URL}, \text{total count} \rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle \text{target}, \text{source} \rangle$ pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle \text{target}, \text{list}(\text{source}) \rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle \text{word}, \text{frequency} \rangle$ pairs. The map function emits a $\langle \text{hostname}, \text{term vector} \rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle \text{hostname}, \text{term vector} \rangle$ pair.

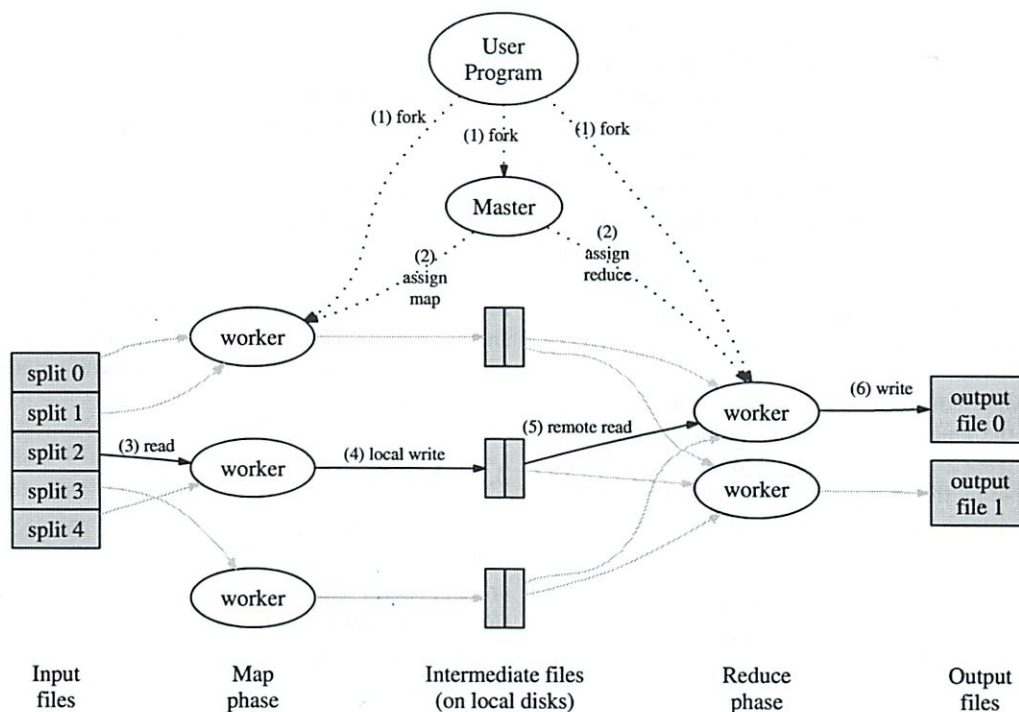


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle \text{word}, \text{document ID} \rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle \text{word}, \text{list}(\text{document ID}) \rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle \text{key}, \text{record} \rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google:

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

- (1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine. *normal*
- (2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
- (3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
- (4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.
- (5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $\text{hash}(\text{key}) \bmod R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

I forget
the examples
from 6.005

pretty
simple...

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker *A* will read the data from worker *B*.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces *R* such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the *R* temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of *R* files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very

easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task *R*₁ is equivalent to the output for *R*₁ produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task *R*₂ may correspond to the output for *R*₂ produced by a different sequential execution of the non-deterministic program.

Consider map task *M* and reduce tasks *R*₁ and *R*₂. Let *e*(*R*_{*i*}) be the execution of *R*_{*i*} that committed (there is exactly one such execution). The weaker semantics arise because *e*(*R*₁) may have read the output produced by one execution of *M* and *e*(*R*₂) may have read the output produced by a different execution of *M*.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies on different machines). The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into *M* pieces and the reduce phase into *R* pieces, as described above. Ideally, *M* and *R* should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large *M* and *R* can be in our implementation, since the master must make *O*(*M* + *R*) scheduling decisions and keeps *O*(*M* * *R*) state in memory as described above. (The constant factors for memory usage are small however: the *O*(*M* * *R*) piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a “straggler”: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on

the intermediate key. A default partitioning function is provided that uses hashing (e.g. “ $hash(key) \bmod R$ ”). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using “ $hash(Hostname(urlkey)) \bmod R$ ” as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form $\langle \text{the}, 1 \rangle$. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, “text”

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal,

the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. *gdb*).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:


```

Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
      EmitIntermediate(w, "1");

```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE

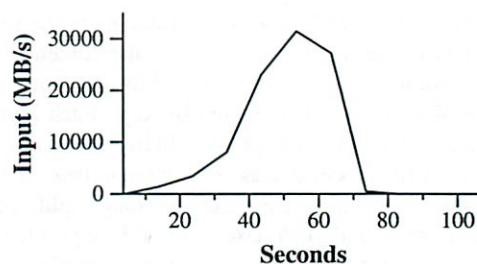


Figure 2: Data transfer rate over time

disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

5.2 Grep

The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the

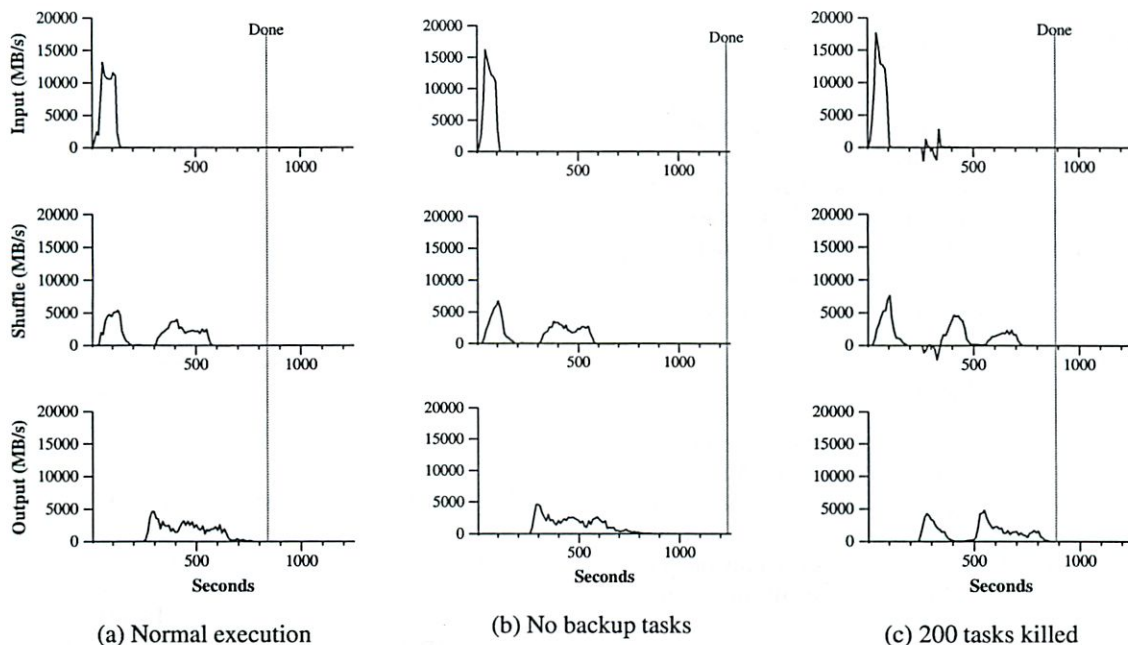


Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This function passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for *grep* had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for

the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

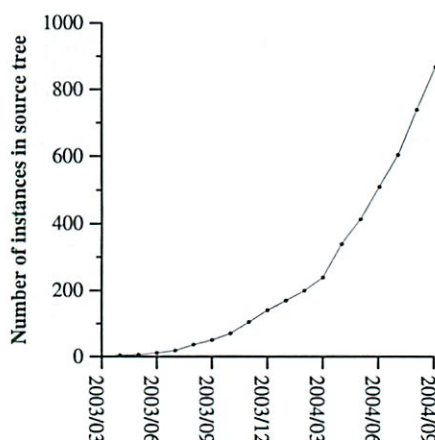


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique <i>map</i> implementations	395
Unique <i>reduce</i> implementations	269
Unique <i>map/reduce</i> combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke,

David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

- [1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.
- [2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.
- [3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
- [4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.
- [5] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.
- [6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.
- [7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.
- [8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that

make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

- [9] S. Gortlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.
- [10] Jim Gray. Sort benchmark home page. <http://research.microsoft.com/barc/SortBenchmark/>.
- [11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
- [12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.
- [13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.
- [14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
- [15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.
- [16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.
- [17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.
- [18] Jim Wylie. Spsort: How to sort a terabyte quickly. <http://almel.almaden.ibm.com/cs/spsort.pdf>.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
                i++;

            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
```

```
        if (start < i)
            Emit(text.substr(start, i-start), "1");
        }
    };
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input(i);
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Specify the output files:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2000
    // machines and 100 MB of memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();

    // Done: 'result' structure contains info
    // about counters, time taken, number of
    // machines used, etc.

    return 0;
}
```

Quiz Fri

MapReduce or Hadoop
open source

- paper more understandable
- many companies use - pretty much all of them
- Twitter: find people w/ similar interests
or X people are interested in motorcycle
or posting on Twitter

me \Rightarrow Followers

Can find the interest of this

Parallel Pre-fix

~~The~~ What he did in grad school

The async version of this

Say we have $a_0 \oplus a_1 \oplus a_2 \oplus a_3 \dots$

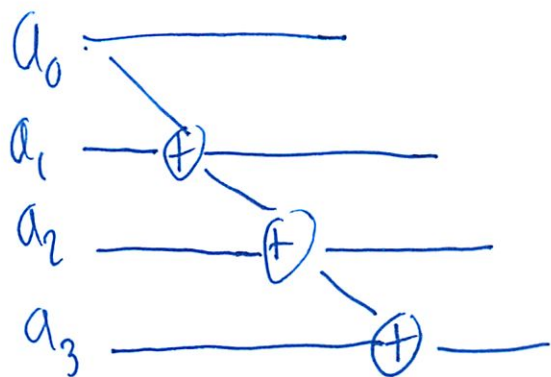
a_0

$a_0 \oplus a_1$

$a_0 \oplus a_1 \oplus a_2$

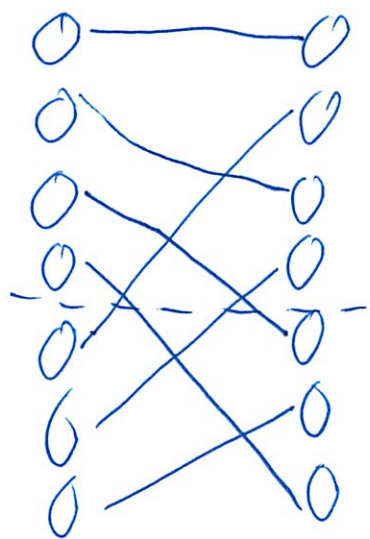
$a_0 \oplus a_1 \oplus a_2 \oplus a_3$

② So do



Or a shuffle exchange pattern

A perfect shuffle



this pattern $\log n$
→ times
gives you
the parallel prefix

Then by having an interesting plus op we
can have all sorts of smart operations

So there ~~was~~ was a lot of work that went into
this 20-30 years ago.

(3)

Playing Map + Reduce

Sometimes ~~better~~ the reduce is more expensive

Mapper does sorting as they process

Here Mapper splits names into 10 sets
by 1st digit

Gives each set to a reducer

Reducer sorts the items in order
by 2nd digit

Now we have a sorted list

If # is IQ

Could ask who has the largest IQ

Or who has IQ b/w 20-40

Wrong: What is Nixon's IQ

- Swap key and value

- hash name 0-9 then give to right group

④

It want to get results back to Mapper
Mapper adds its name to the key value pair

Google query; which page has Nixon on
Gets sent to Mapper

Normal site → SQL query for Nixon

Google waits for a few hundredth seconds
- a few more ~~people~~ people search it

Sends keyword, your mapper name, index to IP requestor
to Reduc - who has a section of alphabet

Bring requests to Reducer

Reducer looks up a result - how many pages
that query is on

Then mapper gets request back

Reducer 1 lookup per query

5

Web crawler does the index

- A saved finished results on Reducer

Multiple Word Queries

- Mapper can send a query for each word to Reducer
- then when returning to Mapper - Mapper finds intersection

This is of course a way over simplified ~~one~~ version

Google only gives 1000 results for each query

Load Balancing - having just right # of Mappers + Reducers

- must split well - balance communication AND computation

Can mirror reducers

- have multiple of the same

(6)

Map Reduce very bad at bandy cases
- so not used for scientific computing

- ie simulating ocean in Cape Cod

- look for neighbors

- but when ocean meets land it's diff

- small places like CapeCod even worse

M.I.T. DEPARTMENT OF EECS

6.033 - Computer System Engineering

Traceroute Hands-On Assignment

Hands-on 3: Internet Routes and Measuring Round Trip Times

Note:

Some students have reported having problems with running the traceroute command on the Debathena cluster machines. If you get the error 'traceroute: Command not found.', run the command

```
sudo aptitude install traceroute
```

to install it for the duration of a login session. Another option is ssh'ing into linux.mit.edu and then running the traceroute command.

Complete the following hands-on assignment. Do the activities described, and hand in the answers to the numbered questions at the **beginning of recitation**. As usual, submit your solutions using the online submission site before recitation.

In this assignment you will get a chance to experiment with two very useful and widely-used network diagnostic tools, traceroute and ping, to expose you to some of the interesting quirks in network routing and packet round trip times.

We recommend, but do not require, that you perform this assignment on Athena. **Part of this assignment cannot be completed on athena.dialup.mit.edu due to security restrictions.** Please note that the TAs cannot guarantee tech support if you do not use an Athena workstation. In either case, please make sure you use a workstation on the MIT network. Some results may be quite different if you use an off-campus network.

0. Measuring Round Trip Times With Ping

In the first two exercises, you will use the ping utility to send echo requests to a number of different hosts. The ping utility is one of the more useful utilities for testing a network. It allows you to measure the time that it takes for a packet to travel through the Internet to a remote host and back. The ping utility works by sending a short message, called an *echo-request*, to a host using the Internet Control Message Protocol (ICMP). A host that supports ICMP (and most do) and receives an echo-request message simply replies by sending an *echo-response* back to the originating host.

In many of the exercises, you will be referring to hosts via their DNS names rather than their IP addresses. (For more information about Internet hostnames and DNS, and how these relate to IP addresses, please see Section 4.4 of the course notes.)

For more information about ping, look at the man page on ping and the specifications for ICMP, located in RFC 792. Section 7.13.4 of the course notes describes ICMP as well.

```
athena% man ping
```

To use the ping command on Athena, run a command such as:

```
athena% ping www.google.com
```

If you run ping from a Sun workstation, you may have to use the `-s` option to get it to display the results that you want. Type `machtype` to determine the type of machine you are using. If you have any more questions, see the man pages for more details on how to use ping.

A. Round Trip Times:

In the following two questions, you are asked to use the ping utility to measure the round trip times to several hosts on

the Internet.

For the following hosts, send 10 packets, each with a length of 56 data bytes. *Note:* You may find that the packet responses are 64 bytes instead of 56 bytes. Look at [RFC 792](#) to find out the reason.

The hosts are:

www.csail.mit.edu
www.berkeley.edu
www.usyd.edu.au
www.kyoto-u.ac.jp

Question 1: Indicate what percentage of packets sent resulted in a successful response. For the packets from which you received a response, write down the minimum, average, and maximum round trip times in milliseconds. Note that ping reports these times to you if you tell it how many packets to send on the command line.

Question 2: Explain the differences in minimum round trip time to each of these hosts.

Question 3: Now send pings with 56, 512 and 1024 byte packets to the 4 hosts above. Write down the minimum, average, and maximum round trip times in milliseconds for each of the 12 pings. Why are the minimum round-trip times to the same hosts different when using 56, 512, and 1024 byte packets?

B. Unanswered Pings:

For the following hosts, send 100 packets that have a length of 56 data bytes. Indicate what percentage of the packets resulted in a successful response.

www.wits.ac.za (University of the Witwatersrand, Johannesburg)
www.microsoft.com

Question 4: For some of the hosts, you may not have received any responses for the packets you sent. What are some reasons as to why you might have not gotten a response? (Be sure to check the hosts in a web browser.)

1. Understanding Internet routes using traceroute

As the name implies, `traceroute` essentially allows you to trace the entire route from your machine to a remote machine. The remote machine can be specified either as a name or as an IP address.

We include a sample output of an execution of `traceroute` and explain the salient features. The command:

```
% traceroute www.google.com
```

tries to determine the path from the source machine (`vinegar-pot.mit.edu`) to `www.google.com`. The machine encountered on the path after the first hop is `NW12-RTR-2-SIPB.MIT.EDU`, the next is `EXTERNAL-RTR-1-BACKBONE-2.MIT.EDU`, and so on. In all, it takes 13 hops to reach `py-in-f99.google.com`. The man page for `traceroute` (`athena% man traceroute`) contains explanations for the remaining fields on each line.

```
% traceroute www.google.com
traceroute: Warning: www.google.com has multiple addresses; using 64.233.167.99
traceroute to www.l.google.com (64.233.167.99), 30 hops max, 40 byte packets
 1  NW12-RTR-2-SIPB.MIT.EDU (18.181.0.1)  0.476 ms  0.318 ms  0.237 ms
 2  EXTERNAL-RTR-1-BACKBONE-2.MIT.EDU (18.168.1.18)  0.827 ms  0.624 ms  0.753 ms
 3  EXTERNAL-RTR-2-BACKBONE.MIT.EDU (18.168.0.27)  1.097 ms  0.772 ms  0.887 ms
 4  207.210.142.233 (207.210.142.233)  0.578 ms  0.549 ms  0.713 ms
 5  207.210.142.1 (207.210.142.1)  0.750 ms  2.530 ms  1.178 ms
 6  207.210.142.2 (207.210.142.2)  5.886 ms  15.387 ms  5.762 ms
 7  64.57.29.21 (64.57.29.21)  24.732 ms  24.693 ms  24.695 ms
 8  72.14.236.215 (72.14.236.215)  31.733 ms  27.588 ms  216.239.49.34 (216.239.49.34)  27.810 ms
 9  66.249.94.235 (66.249.94.235)  12.495 ms  209.85.252.166 (209.85.252.166)  36.961 ms  26.459 ms
10  216.239.46.224 (216.239.46.224)  33.736 ms  33.396 ms  209.85.248.221 (209.85.248.221)  26.130
```



```
11 66.249.94.133 (66.249.94.133) 26.126 ms 72.14.232.53 (72.14.232.53) 25.744 ms 25.611 ms
12 66.249.94.133 (66.249.94.133) 26.183 ms 27.460 ms 72.14.232.70 (72.14.232.70) 37.800 ms
13 py-in-f99.google.com (64.233.167.99) 28.249 ms 26.050 ms 26.398 ms
```

A. Basics:

Question 5:

In at most 50 words, explain how traceroute discovers a path to a remote host. The man page might be useful in answering this question.

B. Routine Asymmetries:

changed since assignment issued

For this exercise, you need to use the traceroute server at <http://www.slac.stanford.edu/cgi-bin/nph-traceroute.pl>. When you view this web page, execute a traceroute (trace) to your machine (run `/sbin/ifconfig` to find the IP Address of your machine). Run your local traceroute to whatever server Stanford tells you it is running traceroute from for this question. **Note:** It is important to run this on an Athena machine. If the Stanford traceroute does not work, or if you get no reply after 2--3 minutes, you should try one of the other looking glass servers on this page: <http://www.traceroute.org/#Looking%20Glass> If you use a different server, make sure that you note in your hands-on that you used a different server than the question asked for.

Now run this on your machine:

```
athena% traceroute www1.slac.stanford.edu
```

Question 6: Show the output of traceroute from each direction above.

Question 7: Describe anything unusual about the output. Are the same routers traversed in both directions? If not, why might this happen?

C. Blackholes:

At the command prompt, type:

```
athena% traceroute 18.31.0.200
```

Question 8: Show the output of the above command. Describe what is strange about the observed output, and why traceroute gives you such an output. Refer to the traceroute man page for useful hints.

2. Border Gateway Protocol (BGP)

For this last question on the topic of Internet routing, you need to refer to the BGP routing table data below. This table shows all of the BGP routing entries that a particular router (near the University of Oregon) refers to when forwarding any packets to MIT (IP Address 18.*.*.*). *prefix*

As described in the Internet routing paper, recall that BGP is a path vector protocol. Each line of this table lists a distinct path from this router to MIT, from which it will choose one to use. The Next Hop field is the IP address of the router that forwards packets for each path listed in the table. The Path field is the list of autonomous systems the path traverses on its way to MIT. The other fields (Metric, LocPrf, Weight) may be used by the router to decide which one of the possible paths to use.

BGP table version is 9993576, local router ID is 198.32.162.100
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
S Stale
Origin codes: i - IGP, e - EGP, ? - incomplete

Network	Next Hop	Metric	LocPrf	Weight	Path
* 18.0.0.0	216.140.8.59	413		0	6395 3356 3 i
*	216.140.2.59	982		0	6395 3356 3 i
*	141.142.12.1			0	1224 22335 11537 10578 3 i

↓ from

↑ direct link

*	209.249.254.19	125	0 6461 3356 3 i
*	202.232.0.2		0 2497 3356 3 i
*	209.10.12.125	8204	0 4513 3356 3 i
*	208.51.113.253		0 3549 174 16631 3 3 3 i
*	209.123.12.51		0 8001 1784 10578 3 i
*	209.10.12.156	0	0 4513 3356 3 i
*	195.66.224.82		0 4513 3356 3 i
*	209.10.12.28	8203	0 4513 3356 3 i
*	203.181.248.233		0 7660 11537 10578 3 i
*	64.50.230.2		0 4181 174 174 174 16631 3 3 3 i
*	195.66.232.254		0 5459 2649 174 174 174 16631 3 3 3 i
*	195.66.232.239		0 5459 2649 174 174 174 16631 3 3 3 i
*	64.50.230.1		0 4181 174 174 174 16631 3 3 3 i
*	194.85.4.55		0 3277 8482 29281 702 701 3356 3 i
*	207.172.6.227	83	0 6079 10578 3 i
*	207.172.6.162	62	0 6079 10578 3 i
*	129.250.0.85	11	0 2914 174 16631 3 3 3 i
*	206.220.240.95		0 10764 11537 10578 3 i
*	217.75.96.60		0 16150 8434 3257 3356 3 i
*	66.185.128.48	514	0 1668 3356 3 i
*	206.24.210.26		0 3561 3356 3 i
*	216.191.65.118		0 15290 174 16631 3 3 3 i
*	216.191.65.126		0 15290 174 16631 3 3 3 i
*	209.161.175.4		0 14608 19029 3356 3 i
*	202.249.2.86		0 7500 2497 3356 3 i
*	208.186.154.35	0	0 5650 3356 3 i
*	167.142.3.6		0 5056 1239 3356 3 i
*	64.200.151.12		0 7911 3356 3 i
*	195.219.96.239		0 6453 3356 3 i
*	208.186.154.36	0	0 5650 3356 3 i
*	203.194.0.12		0 9942 16631 174 174 174 16631 3 3 3 i
*	213.200.87.254	40	0 3257 3356 3 i
*	216.218.252.145		0 6939 3356 3 i
*	216.18.63.137		0 6539 174 16631 3 3 3 i
*	216.218.252.152		0 6939 3356 3 i
*	195.249.0.135		0 3292 3356 3 i
*	65.106.7.139	3	0 2828 174 16631 3 3 3 i
*	207.45.223.244		0 6453 3356 3 i
*	207.246.129.14		0 11608 6461 3356 3 i
*	207.46.32.32		0 8075 174 16631 3 3 3 i
*	129.250.0.11	0	0 2914 174 16631 3 3 3 i
*	134.55.200.1		0 293 11537 10578 3 i
*	193.0.0.56		0 3333 3356 3 i
*	216.140.14.186	3	0 6395 3356 3 i
*	198.32.8.196	960	0 11537 10578 3 i
*	64.200.95.239		0 7911 3356 3 i
*	196.7.106.245		0 2905 701 3356 3 i
*	154.11.63.86		0 852 174 16631 3 3 3 i
*	134.222.85.45	0	0 286 209 3356 3 i
*	213.140.32.146		0 12956 174 16631 3 3 3 i
*	164.128.32.11		0 3303 3356 3 i
*	213.248.83.240		0 1299 3356 3 i
*	154.11.98.18		0 852 174 16631 3 3 3 i
*>	4.68.0.243	0	0 3356 3 i
*	204.42.253.253	0	0 267 2914 174 16631 3 3 3 i
*	206.186.255.223		0 2493 3602 174 16631 3 3 3 i
*	193.251.128.22		0 5511 3356 3 i
*	203.62.252.26		0 1221 4637 3356 3 i
*	12.0.1.63		0 7018 3356 3 i
*	144.228.241.81 4294967294		0 1239 3356 3 i

is this the best or lying?

Question 9: From the path entry data, which Autonomous System (AS) number corresponds to MIT?

Question 10: What are the Autonomous System (AS) numbers of each AS which advertises a direct link to MIT?

If you'd like to explore BGP and Internet routing in more depth, you may wish to take take 6.829 Computer Networks.

I should be a practice question

Ping ^{echo}
~~each~~ request
ICMP

Oh cool mach \Rightarrow type

Ping some hosts

- C = Count
- S = size

(being to copy all this data)

What is the max packet size

↳ MTU

Can use Path MTU discovery

Trace route

Explain how it works...

②

56 words!

think that is ok

• BGP

next hops

metric or a local pref

So a bunch of choices

Quiz 1

Mean 69.97

Median 70

My score = 70

This unit: networking

Before: Complexity

Modularity

Client/Server

~~Abstraction~~

Naming

On 1 PC \rightarrow OS

Performance

Client-Server using Network

- more enforced modularity
- lets us share ind of where we are

Networks are a systems tool!

- are a system themselves

②

Main Problems

This class a more systems class
than 6.02

Economical

- Sharing + Utilization

Organizational

- Routing
- end to end

Physical

- Speed of light

What is the interface to the Network?

- Actually has pretty weak guarantees

Value of 1 network:

↳ Universality

incentive to have 1 network

Value of network $O(n^2)$

- value \propto as more people join

(3) Network is like a black box to applications
↳ but should it be?
Smart vs dumb?
↑ the POTS ↑ the Internet

key challenges: standardization of protocols

How is the network organized topographically?

- n^2 connections
- hub / dispatcher: - reduces to n connections
- but critical to reliability + performance

Routing

Hard to do at Internet scale

Global problem → platform ID?

17.26.49 → location hints

Sharing a link

How multiplex a link over ~~convers~~

Or divide links into time slots TDM

(4)

- 'isochronous' connection
 - This is how phone worked
 - you had a guaranteed link
 - AT&T made sure enough bandwidths
- But on data network - may do nothing \rightarrow asynchronous
So multiplex
- But if overload (more in than out)
 - if permanent: bigger link required
 - medium state: do you send a message back to sender?
no - just drop a package!
- Sidebar TCP does not work that well on wireless
- Short-term: buffer packets
- Cable Modems have large buffers
 - ACM: buffer bloat
 - kills latency (\uparrow latency)

5

Utilization

$$C = \sum_i \text{max send rate}$$

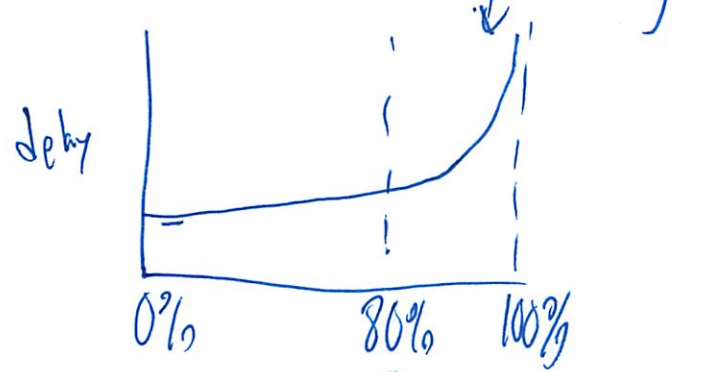
← expensive
This is usually \gg avg
So instead plan on something much smaller
called statistical multiplexing

← Users actually spread around their bursts

Can see network graphs of MIT's network
- no difference on Sat/Sun to weekday

What util to run link at?

100% - efficiently used
but lots of Buffering



↑ So go around here

MIT 50-60%

6

Errors

- link cabling
 - back hoses cut link
- switch dies
- operator error
- wrong BGP protocol outputted

* Have end host deal w/ most of this

Speed of Light

- its not fast enough !! :(
- 1 ft / nano second
- east to west coast = 14 ms
- if you waited for a read receipt
 - ↳ only 36 bytes/sec
- so implement complex transmission control protocols
- setting your sliding window
- large discrepancy in speed of network

⑦

↓ technology
↓ time

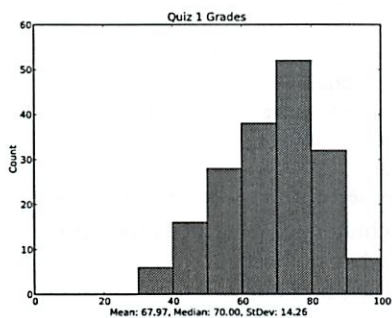
The ~~Internet~~ Internet is a Best Effort network

traceroute - a shows ya AS

Internet's best effort

Hosts add best-effort protocols on top of (TCP)

Quiz 1 histogram



L10: Network Systems

Frans Kaashoek

6.033 Spring 2012

<http://web.mit.edu/6.033>

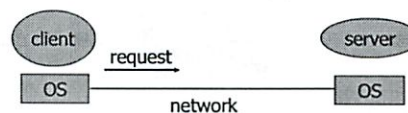
Some slides are from lectures by
 Nick Mckeown, Ion Stoica, Dina
 Katabi, Hari Balakrishnan, Sam
 Madden, and Robert Morris



What have you seen so far?

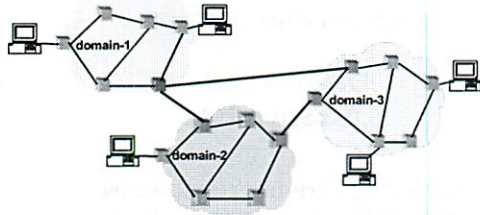
Systems	Complexity Modularity Dtechnology/dt	Hierarchy Therac-25
Client/service design	Enforced modularity	X windows
Naming systems	Gluing systems	File system name space/DNS
Operating systems	Client/service with in a computer	Eraser and Unix
Performance	Coping with bottlenecks	MapReduce

Client/service using network



- Sharing irrespective of geography
- Strong modularity through geographic separation

Network is a system too!



- Network consists of many networks, many links, many switches
- Internet is a case study of successful network system

Today's topic: problems and approach

- Economical:
 - Universality
 - Topology, Sharing, Utilization
- Organizational
 - Routing, Addressing, Packets, Delay
 - Best-effort contract
- Physical
 - Errors, speed of light, wide-range of parameters

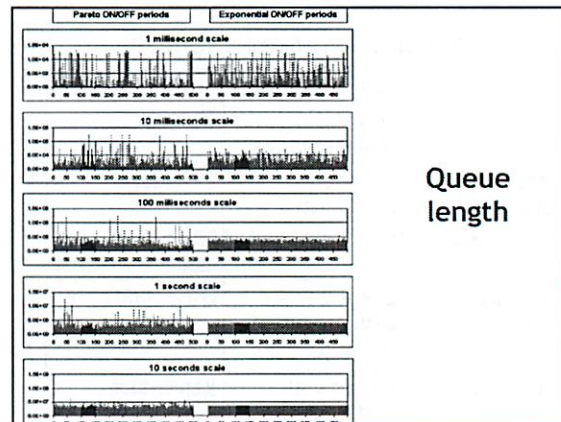
Design challenge: what does the network do and what do hosts do?

- Internet: best-effort

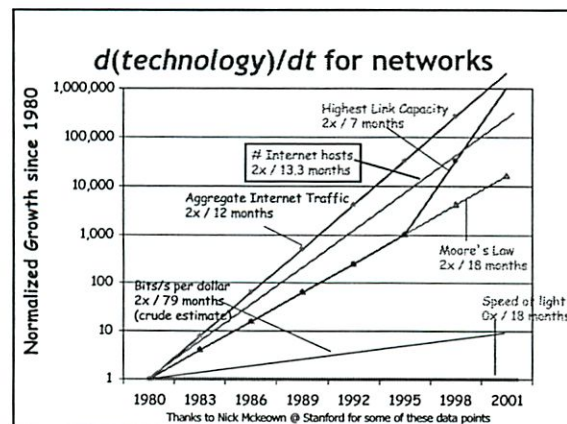
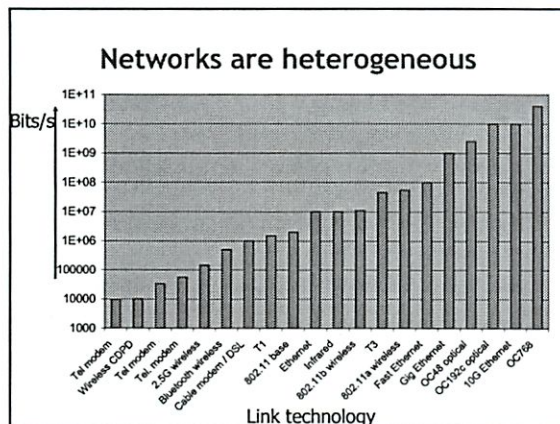
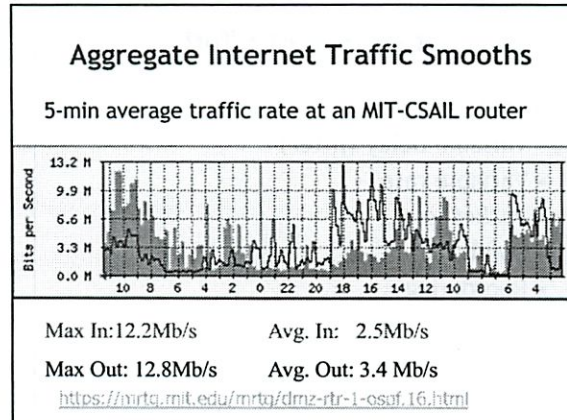
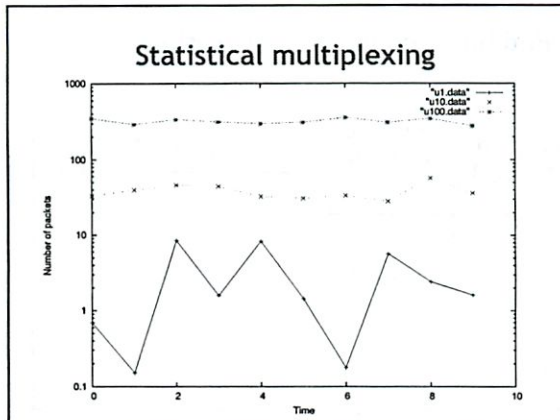
Asynchronous Multiplexing/ Demultiplexing



- Multiplex using a queue
 - Switch need memory/buffer
- Demultiplex using information in packet header
 - Header has destination
 - Switch has a forwarding table that contains information about which link to use to reach a destination



Queue length

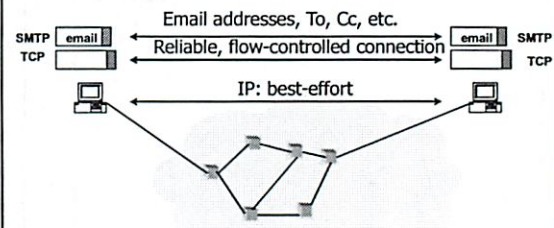


Internet: Best Effort

No Guarantees:

- Variable Delay (jitter)
- Variable rate
- Packet loss
- Duplicates
- Reordering

End hosts implement everything else



3/12

6.033 2011 Lecture 10: Networking Intro (6.02 in a single lecture)

First of 4 lectures on data networks.

Overview today: identify the problems and approach

Dig into details over next 3 lectures.

- * Build on 6.02

- * network are a useful systems building block

- * internal workings are a good case study of system design.

Internet in particular an example of a very successful system.

complex enough to be a subject of science, like the weather

What is the goal of data networking?

Allow us to build systems out of multiple computers.

Your problem may be too big for one computer.

You may want multiple computers for structural reasons: client/server.
more fundamental reason:

A key use of computers is for communication between people, orgs.

People are geographically distributed, along with their computers.

So we're forced to deal with inter-computer comms.

System picture:

Hosts. Maybe millions of them. All over the world.

[No cloud yet, but leave room. circular hosts.]

What are the key design problems to be solved?

I'll classify into three types:

O: Organizational, to help human designers cope.

E: Economic, to save operators money.

P: Physics.

E: universality.

Want one net, many uses

Rather than lots of small incompatible/disconnected network worlds.

e.g. Fedex builds its own data net for its customers to track packages

private nets sometimes good for reliability, predictable service, security

The more people that use a network, the more useful it is for everybody.

Value to me is proportional to N people using the same net.

Value to society is proportional to N^2 .

What technical tools help us achieve universality?

Standard protocols make it easy to build systems.

But don't want to prevent evolution by freezing design w/ standards.

Hard: standardize just what's required to make net generally useful,

but not the things that might need to be changed later.

"Dumb" versus "smart" network?

One universal net means it's *not* part of each system design.

It's a black box; simplifies design of systems that use it.

Symbolically, a cloud that hides internal workings.

[draw network cloud]

E: topology.

[three new pictures; circular hosts, square switches]

Look inside the cloud.

Wire between every pair?

pro: network (wires) is transparent, passes analog signals.

It's never busy. Never any question of how to reach someone.

con: host electronics. laying wires expensive.

Star network. "switch". Federal Express in Memphis.

pro: less wire cost. easy to find a path.

con: load and cost and unreliability of hub.

Mesh topology. Routers and links.

pro: low wire cost. limited router fan-out. some path redundancy.

con: shared links, higher load. complex routing.

O: routing.

Find paths given dst "address".

Harder in mesh than in star or hierarchy.

[diagram: two available paths]

change far away may require re-routing
-> has a global element, thus scaling problems

many goals:

Efficiency: low delay, fast links
Spread load
Adapt to failures
Minimize payments
Cope w/ huge scale

Routing is a key problem.

O: addressing.

Hosts need to be able to specify destination to network.
specifically, the address you give to the routing system
18.7.22.69, not web.mit.edu

ideal:

every host has a unique ID, perhaps 128 bits assigned at random
routing system can find address, wherever it is

so i can connect to my PDA, whether i left it at home or in the office

no-one knows how to build a routing system for large #s of flat addresses!

maybe can layer on top of something else, but not directly

In practice, encode location hints in addresses.

hierarcical addresses, e.g. 18.7.whatever

[diagram: 18 area, 18.7 area, ...]

rest of inet knows only 18, not every host in 18

Trouble if hosts move or topology changes.

hard to exploit non-hierarchical links (rest of Inet can't use MIT-Harvard)

routing and addressing often very intertwined

E: sharing.

Must multiplex many conversations on each physical wire.

1. how to multiplex?

A. isochronous.

[input links, router1, router2, link, repeating cycle]

reserved b/w, predictable delay, originally designed for voice
called TDM

B. asynchronous

data traffic tends to be bursty - not evenly spaced

you think, then click on a link that loads lots of big images

wasteful to reserve b/w for a conversation

so send and forward whenever data is ready

[diagram: input links, router1, link, router2, output links]

divide data into packets, each with header w/ info abt dst

2. how to keep track of conversations?

A. connections

like phone system

you tell network who you want to talk to

figures out path in advance, then you talk

required for isochronous traffic

can allow control of load balance for async traffic

maybe allow smaller packet headers (just small connection ID)

connection setup complex, forwarding simpler

B. connectionless / datagrams

many apps don't match net-level connections, e.g. DNS lookups

each packet self-contained, treated separately

packet contains full src and dst addresses

each may take a different route through network!

shifts burden from network to end hosts (connection abstraction)

E: Overload

more demand than capacity

consequence of sharing and growth and bursty sources

how does it appear?

isochronous net:

new calls blocked, existing calls undisturbed

makes sense if apps have constant unchangeable b/w requirements

async net:

[diagram: router with many inputs]

overload is inside the net, not apparent to sending hosts.

net must do something with excess traffic
 depends on time scale of demand > capacity
 very long: buy faster links
 short: tell senders to slow down
 feedback
 elastic applications (file xfer, video with parameterized compression)
 can always add more users, just gets slower for everyone
 often better than blocking calls
 very short: don't drop -- buffer packets -- "queuing"
 demand fluctuates over time
 [graph: varying demand, line for fixed link capacity]
 buffers allow you to delay peaks into valleys
 but only works if valley comes along pretty quick!
 adds complexity. must drop if overload too much. source of most Inet loss.
 Behavior with overload is key.
 [Graph of in rate vs out rate, knee, collapse.]
 Collapse: resources consumed coping with overload.
 This is an important and hard topic: congestion control

E: high utilization.

How much expensive capacity do we have to pay for?
 Capacity \geq peak demand would keep users happy.
 What would that mean?
 single-user traffic is bursty!
 [typical user time vs load graph -- avg low, peak high]
 Worst case is 100x average, so network would be 99% wasted!
 e.g. my one-megabit DSL line is almost entirely idle
 too expensive in core to buy $N_{\text{users}} \times \text{peak}$
 But when you aggregate async users -- peaks and valleys cancel.
 [a few graphs, w/ more and more users, smoother and smoother]
 peak gets closer and closer to average
 100 users in \ll 100x capacity needed for 1 user.
 less and less idle capacity, lower cost per bit transmitted
 "Statistical multiplexing."
 Hugely important to economics of Internet
 Assumes independent users
 mostly true -- but day/night, flash crowds
<https://mrtg.mit.edu/mrtg/dmz-rtr-1-ospf.16.html> (MIT DMZ router to campus backbone)
 50% average daytime utilization maybe typical on core links

Once you buy capacity, you want to get as close to 100% as possible.
 fixed cost, want to maximize revenue/work
 may need feedback to tell senders to speed up!
 want to stay at first knee of in/out graph.
 Or maybe less to limit delay
 [graph: load vs queuing delay]

P: errors and failures.

Communication wires are analog, suffer from noise.
 Backhoes, unreliable control computers.
 General fix: detect, send redundant info (rxmt), or re-route.
 How to divide responsibility?
 You pay your telecommunications provider, so they better be perfect?
 Leads to complex, expensive networks: each piece 100% reliable.
 You don't trust the network, so you detect and fix problems yourself.
 Leads to complex host software.
 Decision of smart network vs smart hosts turns out to be very important.
 Locus of complexity pays costs but also has power.
 "Best design" depends on whether you are operator or user.

P: speed of light.

foot per nanosecond.
 14 ms coast to coast, maybe 40 ms w/ electronics.
 Request/response delay. Byte per rtt \rightarrow 36 bytes/second.
 [picture of pkt going over wire -- and idle wire]
 Need to be more efficient.
 Worse: delay hurts control algorithms. Change during delay.
 Example: Slow down / speed up.

Result: Oscillation between overload and wastefully idle.

O: wide range of adaptability.

One design, many situations.

Applications: file xfer, games, voice, video.

Delay: machine room vs satellite.

Link bit rate: modem vs fiber optics.

Management: a few users, whole Internet.

Solution: Adaptive mechanisms.

Internet in design space

Asynchronous (no reservations)

Packets (no connections)

No b/w or delay guarantees

May drop, duplicate, re-order, or corrupt packets -- and often does

Not of much direct use! End hosts must fix

but gives host flexibility, room for innovation

"best effort"

demo: ping -f

Flood ping. Outputs packets as fast as they come back or one hundred times per second, whichever is more. For every ECHO_REQUEST sent a period ``.''' is printed, while for every ECHO_REPLY received a backspace is printed. This provides a rapid display of how many packets are being dropped.

ssh amsterdam

su

ping -f localhost

ping -f web.mit.edu

ping -f yahoo.com

demo: traceroute -a yahoo.com

6.033: Computer Systems Engineering

Spring 2012

[Home / News](#)[Schedule](#)[Submissions](#)

[General Information](#)[Staff List](#)[Recitations](#)[TA Office Hours](#)

[Discussion / feedback](#)[FAQ](#)[Class Notes Errata](#)[Excellent Writing Examples](#)

[2011 Home](#)

Preparation for Recitation 10

as opposed to best effort?

Read [End-to-end Arguments in System Design](#). As you read this paper, think about the following questions:

- How does the end-to-end argument apply to other systems you know about, for example, the X windows system?
- How does the packet voice example make use of the end-to-end argument?
- Is it OK for lower levels to do a bad job? Why should they do error detection and correction?

Also read:

- The introduction and section 1 of the [Wikipedia article on Network Address Translators](#).
- The [Speak Freely End of Life Announcement](#). Speak Freely was an early voice chat program for the Internet.

After reading these articles, think about the relationship between the end-to-end argument and NATs.

Read 3/11

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

[Top](#) // [6.033 home](#) //

End-to-end principle

From Wikipedia, the free encyclopedia

The **end-to-end principle** is one of the classic design principles of computer networking.^[nb 1] First explicitly articulated in a 1981 conference paper by Saltzer, Reed, and Clark,^{[Ref 1] [nb 2]} it has inspired and informed many subsequent debates on the proper distribution of functions in the Internet and communication networks more generally.

Oh only the end points smart - middle dumb
The end-to-end principle states that application specific functions ought to reside in the *end hosts* of a network rather than in *intermediary nodes* – provided they can be implemented "completely and correctly" in the end hosts. Going back to Baran's work on obtaining reliability from unreliable parts in the early 1960s, the basic intuition behind the original principle is that the payoffs from adding functions to the *network* quickly diminish, especially in those cases where the *end hosts* will have to implement functions for reasons of "completeness and correctness" anyway (regardless of the efforts of the network).^[nb 3]

The canonical example for the end-to-end principle is that of arbitrarily reliable data transfer between two communication end points in a distributed network of nontrivial size, for the only way two end points can obtain perfect reliability is by positive end-to-end acknowledgments plus retransmissions in their absence. In debates about *network neutrality* a common interpretation of the end-to-end principle is that it implies a neutral or "dumb" network.

Contents

- 1 Basic content of the principle
- 2 History
 - 2.1 The basic notion: reliability from unreliable parts
 - 2.2 Early trade-offs: experiences in the Arpanet
 - 2.3 The canonical case: TCP/IP
- 3 Limitations of the principle
- 4 Notes
- 5 References
- 6 External links

Basic content of the principle

The fundamental notion behind the end-to-end principle is that for two processes communicating with each other via some communication means the *reliability* obtained from that means cannot be expected to be perfectly aligned with the *reliability requirements of the processes*. In particular, meeting or exceeding very high reliability requirements of communicating processes separated by networks of nontrivial size is more costly than obtaining the required degree of reliability by positive end-to-end acknowledgements and retransmissions (referred to as PAR or ARQ).^[nb 4] Put differently, it is far easier and more tractable to obtain reliability beyond a certain margin by mechanisms in the *end hosts* of a network rather than in the *intermediary nodes*,^[nb 5] especially when the latter are beyond the control of and accountability to the

former.^[nb 6] An end-to-end PAR protocol with infinite retries can obtain arbitrarily high reliability from any network with a higher than zero probability of successfully transmitting data from one end to another.^[nb 7]

The end-to-end principle does not trivially extend to functions beyond end-to-end error control and correction. E.g., no straightforward end-to-end arguments can be made for communication parameters such as latency and throughput. Based on a personal communication with Saltzer (lead author of the original end-to-end paper^[Ref 2]) Blumenthal and Clark in a 2001 paper note:^[Ref 6]

[F]rom the beginning, the end-to-end arguments revolved around requirements that could be implemented correctly at the end-points; if implementation inside the network is the only way to accomplish the requirement, then an end-to-end argument isn't appropriate in the first place. (p. 80)

History

The meaning of the end-to-end principle has been continuously reinterpreted ever since its initial articulation. Also, noteworthy formulations of the end-to-end principle can be found prior to the seminal 1981 Saltzer, Reed, and Clark paper.^[Ref 2]

The basic notion: reliability from unreliable parts

In the 1960s Paul Baran and Donald Davies in their pre-Arpanet elaborations of networking made brief comments about reliability that capture the essence of the later end-to-end principle. To quote from a 1964 Baran paper:^[Ref 7]

Reliability and raw error rates are secondary. The network must be built with the expectation of heavy damage anyway. Powerful error removal methods exist. (p. 5)

Similarly, Davies notes on end-to-end error control:^[Ref 8] *don't rely on the network*

It is thought that all users of the network will provide themselves with some kind of error control and that without difficulty this could be made to show up a missing packet. Because of this, loss of packets, if it is sufficiently rare, can be tolerated. (p. 2.3)

Early trade-offs: experiences in the Arpanet

The Arpanet was the first large-scale general-purpose packet switching network – implementing several of the basic notions previously touched on by Baran and Davies, and demonstrating a number of important aspects to the end-to-end principle:

Packet switching pushes some logical functions toward the communication end points

If the basic premise of a distributed network is packet switching, then functions such as reordering and duplicate detection inevitably have to be implemented at the logical end points of such network. Consequently, the Arpanet featured two distinct levels of functionality – (1) a lower level concerned with transporting data packets between neighboring network nodes (called IMPs), and (2) a higher level concerned with various end-to-end aspects of the data transmission.^[nb 8] Dave Clark, one of the authors of the end-to-end principle paper, concludes:^[Ref 11] "The discovery of packets is not a

consequence of the end-to-end argument. It is the success of packets that make the end-to-end argument relevant" (slide 31).

No arbitrarily reliable data transfer without end-to-end acknowledgment and retransmission mechanisms

The Arpanet was designed to provide reliable data transport between any two end points of the network – much like a simple I/O channel between a computer and a nearby peripheral device.^[nb 9]

In order to remedy any potential failures of packet transmission normal Arpanet messages were handed from one node to the next node with a positive acknowledgment and retransmission scheme; after a successful handover they were then discarded,^[nb 10] no source to destination retransmission in

case of packet loss was catered for. However, in spite of significant efforts, perfect reliability as envisaged in the initial Arpanet specification turned out to be impossible to provide – a reality that became increasingly obvious once the Arpanet grew well beyond its initial four node topology.^[nb 11]

The Arpanet thus provided a strong case for the inherent limits of network based hop-by-hop reliability mechanisms in pursuit of true end-to-end reliability.^[nb 12]

There network responsible for hops

Trade-off between reliability, latency, and throughput

The pursuit of perfect reliability may hurt other relevant parameters of a data transmission – most importantly latency and throughput. This is particularly important for applications that require no perfect reliability, but rather value predictable throughput and low latency – the classic example being interactive real-time voice applications. This use case was catered for in the Arpanet by providing a raw message service that dispensed with various reliability measures so as to provide faster and lower latency data transmission service to the end hosts.^[nb 13]

had

The canonical case: TCP/IP

To this day the Internet is characterized mainly by the primacy of the IP protocol – providing a connectionless datagram service with no delivery guarantees and effectively no QoS parameters – at the narrow waist of the hourglass abstraction of the Internet architecture. Arbitrary protocols may sit on top of IP; however, TCP has been the one most widely used, given that it provides a reliable end-to-end transport service to end points thus communicating. The functional separation between IP and TCP serves at the canonical exemplification of the end-to-end principle and is often used in a normative sense when lamenting violations of network neutrality.

Limitations of the principle

The most important limitation of the end-to-end principle is that its basic conclusion – put functions in the application end points rather than the intermediary nodes – is not trivial to operationalize. Specifically:

- it assumes a notion of distinct application end points as opposed to intermediary nodes that makes little sense when considering the structure of distributed applications;
- it assumes a dichotomy between non-application-specific and application-specific functions (the former to be part of the operations between application end points and the latter to be implemented by the application end points themselves) while arguably no function to be performed in a network is fully orthogonal to all possible application needs; *yeah where is that line*
- it remains silent on functions that may not be implemented "completely and correctly" in the application end points and places no upper bound on the amount of application specific functions that may be placed with intermediary nodes on grounds of performancy considerations, economic trade-offs, etc.

diff ways to look at

Notes

1. ^ See Denning's Great Principles of Computing.
2. ^ The 1981 paper^[Ref 1] was published in ACM's TOCS in an updated version in 1984.^[Ref 2] Also, there is a 1980 predecessor version with the same title, published as an internal note at MIT's Laboratory for Computer Science.^[Ref 3]
3. ^ The full quote from the Saltzer, Reed, Clark paper reads:^[Ref 2]

In a system that includes communications, one usually draws a modular boundary around the communication subsystem and defines a firm interface between it and the rest of the system. When doing so, it becomes apparent that there is a list of functions each of which might be implemented in any of several ways: by the communication subsystem, by its client, as a joint venture, or perhaps redundantly, each doing its own version. In reasoning about this choice, the requirements of the application provide the basis for the following class of arguments: The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.) We call this line of reasoning against low-level function implementation the end-to-end argument. (p. 278)
4. ^ In fact, even in local area networks there is a non-zero probability of communication failure – "attention to reliability at higher levels is required regardless of the control strategy of the network".^[Ref 4]
5. ^ Put in economics terms, the marginal cost of additional reliability in the network exceeds the marginal cost of obtaining the same additional reliability by measures in the end hosts. The economically efficient level of reliability improvement inside the network depends on the specific circumstances; however, it is certainly nowhere near zero.^[Ref 2]

Clearly, some effort at the lower levels to improve network reliability can have a significant effect on application performance. (p. 281)
6. ^ The possibility of enforceable contractual remedies notwithstanding, it is impossible for any network in which intermediary resources are shared in a non-deterministic fashion to guarantee perfect reliability. At most, it may quote statistical performance averages.
7. ^ More precisely:^[Ref 5]

A correctly functioning PAR protocol with infinite retry count never loses or duplicates messages. [Corollary:] A correctly functioning PAR protocol with finite retry count never loses or duplicates messages, and the probability of failing to deliver a message can be made arbitrarily small by the sender. (p. 3)
8. ^ In accordance with the Arpanet RFQ^[Ref 9] (pp. 47 f.) the Arpanet conceptually separated certain functions. As BBN point out in a 1977 paper^[Ref 10]:

[T]he ARPA Network implementation uses the technique of breaking messages into packets to minimize the delay seen for long transmissions over many hops. The ARPA Network implementation also allows several messages to be in transit simultaneously between a given pair of Hosts. However, the several messages and the packets within the messages may arrive at the destination IMP out of order, and in the event of a broken IMP or line, there may be duplicates. The task of the ARPA Network source-to-destination transmission procedure is to reorder packets and messages at their destination, to cull duplicates, and after all the packets of a message have arrived, pass the message on to the destination Host and return an end-to-end acknowledgment. (p. 284)
9. ^ This requirement was spelled out in the Arpanet RFQ^[Ref 9]:

From the point of view of the ARPA contractors as users of the network, the communication

subnet is a self-contained facility whose software and hardware is maintained by the network contractor. In designing Interconnection Software we should only need to use the I/O conventions for moving data into and out of the subnet and not otherwise be involved in the details of subnet operation. Specifically, error checking, fault detection, message switching, fault recovery, line switching, carrier failures and carrier quality assessment, as required to guarantee reliable network performance, are the sole responsibility of the network contractor. (p. 25)

10. ^ Notes Walden in a 1972 paper:^[Ref 12]

Each IMP holds on to a packet until it gets a positive acknowledgment from the next IMP down the line that the packet has been properly received. It is gets the acknowledgment, all is well; the IMP knows that the next IMP now has responsibility for the packet and the transmitting IMP can discard its copy of the packet. (p. 11)

11. ^ By 1973, BBN acknowledged that the initial aim of perfect reliability inside the Arpanet was not achievable:^[Ref 13]

Initially, it was thought that the only components in the network design that were prone to errors were the communications circuits, and the modem interfaces in the IMPs are equipped with a CRC checksum to detect "almost all" such errors. The rest of the system, including Host interfaces, IMP processors, memories, and interfaces, were all considered to be error-free. We have had to re-evaluate this position in the light of our experience. (p. 1)

In fact, as Metcalfe summarizes by 1973,^[Ref 14] "there have been enough bits in error in the Arpanet to fill this quota [one undetected transmission bit error per year] for centuries" (p. 7-28). See also BBN Report 2816 (pp. 10 ff.)^[Ref 15] for additional elaboration about the experiences gained in the first years of operating the Arpanet.

12. ^ Incidentally, the Arpanet also provides a good case for the trade-offs between the cost of end-to-end reliability mechanisms versus the benefits to be obtained thusly. Note that true end-to-end reliability mechanisms would have been prohibitively costly at the time, given that the specification held that there could be up to 8 host level messages in flight at the same time between two end points, each having a maximum of more than 8000 bits. The amount of memory that would have been required to keep copies of all those data for possible retransmission in case no acknowledgment came from the destination IMP was too expensive to be worthwhile. As for host based end-to-end reliability mechanisms – those would have added considerable complexity to the common host level protocol (Host-Host Protocol). While the desirability of host-host reliability mechanisms was articulated in RFC 1, after some discussion they were dispensed with (although higher level protocols or applications were, of course, free to implement such mechanisms themselves). For a recount of the debate at the time see Bärwolff 2010,^[Ref 16] pp. 56-58 and the notes therein, especially notes 151 and 163.
13. ^ Early experiments with packet voice date back to 1971, and by 1972 more formal ARPA research on the subject commenced. As documented in RFC 660 (p. 2),^[Ref 17] in 1974 BBN introduced the raw message service (Raw Message Interface, RMI) to the Arpanet, primarily in order to allow hosts to experiment with packet voice applications, but also acknowledging the use of such facility in view of possibly internetwork communication (cf. a BBN Report 2913^[Ref 18] at pp. 55 f.). See also Bärwolff 2010,^[Ref 16] pp. 80-84 and the copious notes therein.

References

- ^{a b} Saltzer, J. H., D. P. Reed, and D. D. Clark (1981) "End-to-End Arguments in System Design". In: Proceedings of the Second International Conference on Distributed Computing Systems. Paris, France. April 8–10, 1981. IEEE Computer Society, pp. 509-512.
- ^{a b c d e} Saltzer, J. H., D. P. Reed, and D. D. Clark (1984) "End-to-End Arguments in System Design". In: ACM Transactions on Computer Systems 2.4, pp. 277-288. (See also here (<http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf>) for a version from Saltzer's MIT homepage.)
- ^a Saltzer, J. H. (1980). End-to-End Arguments in System Design. Request for Comments No. 185, MIT

- Laboratory for Computer Science, Computer Systems Research Division. (Online copy (<http://web.mit.edu/Saltzer/www/publications/rfc/csr-rfc-185.pdf>)).
4. ^ Clark, D. D., K. T. Pogran, and D. P. Reed (1978). "An Introduction to Local Area Networks". In: *Proceedings of the IEEE* 66.11, pp. 1497–1517.
 5. ^ Sunshine, C. A. (1975). *Issues in Communication Protocol Design – Formal Correctness*. Draft. INWG Protocol Note 5. IFIP WG 6.1 (INWG). (Copy from CBI (<http://xn--brwolff-5wa.de/public/Sunshine-1975-Issues-in-Communication-Protocol-Design--Formal-Correctness--INWG-Note-5.pdf>)).
 6. ^ Blumenthal, M. S. and D. D. Clark (2001). "Rethinking the Design of the Internet: The End-to-End Arguments vs. the Brave New World". In: *ACM Transactions on Internet Technology* 1.1, pp. 70–109. (Online pre-publication version (<http://mia.ece.uic.edu/~papers/Networking/pdf00002.pdf>)).
 7. ^ Baran, P. (1964). "On Distributed Communications Networks". In: *IEEE Transactions on Communications* 12.1, pp. 1–9.
 8. ^ Davies, D. W., K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson (1967). "A Digital Communication Network for Computers Giving Rapid Response at Remote Terminals". In: *SOSP '67: Proceedings of the First ACM Symposium on Operating System Principles*. Gatlinburg, TN. October 1–4, 1967. New York, NY: ACM, pp. 2.1–2.17.
 9. ^ ^a ^b Scheblik, T. J., D. B. Dawkins, and Advanced Research Projects Agency (1968). RFQ for ARPA Computer Network. Request for Quotations. Advanced Research Projects Agency (ARPA), Department of Defense (DoD). (Online copy (http://www.cs.utexas.edu/users/chris/DIGITAL_ARCHIVE/ARPANET/RFQ-ARPA-IMP.pdf)).
 10. ^ McQuillan, J. M. and D. C. Walden (1977). "The ARPA Network Design Decisions". In: *Computer Networks* 1.5, pp. 243–289. (Online copy (<http://www.walden-family.com/public/whole-paper.pdf>)). Based on a Crowther et al. (1975) paper, which is based on BBN Report 2918, which in turn is an extract from BBN Report 2913, both from 1974.
 11. ^ Clark, D. D. (2007). *Application Design and the End-to-End Arguments*. MIT Communications Futures Program Bi-Annual Meeting. Philadelphia, PA. May 30–31, 2007. Presentation slides. (Online copy (<http://cfp.mit.edu/events/may07/presentations/CLARK%20Application%20Design.ppt>)).
 12. ^ Walden, D. C. (1972). "The Interface Message Processor, Its Algorithms, and Their Implementation". In: *AFCET Journées d'Études: Réseaux de Calculateurs (AFCET Workshop on Computer Networks)*. Paris, France. May 25–26, 1972. Association Française pour la Cybernétique Économique et Technique (AFCET). (Online copy (<http://www.walden-family.com/public/1972-afcet-paris.pdf>)).
 13. ^ McQuillan, J. M. (1973). *Software Checksumming in the IMP and Network Reliability*. RFC 528. Historic. NWG.
 14. ^ Metcalfe, R. M. (1973). "Packet Communication". PhD thesis. Cambridge, MA: Harvard University. Online copy (<http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-114.pdf>) (revised edition, published as MIT Laboratory for Computer Science Technical Report 114). Mostly written at MIT Project MAC and Xerox PARC.
 15. ^ Bolt, Beranek and Newman Inc. (1974). *Interface Message Processors for the Arpa Computer Network*. BBN Report 2816. Quarterly Technical Report No.5, 1 January 1974 to 31 March 1974. Bolt, Beranek and Newman Inc. (BBN). (Private copy, courtesy of BBN (<http://xn--brwolff-5wa.de/public/BBN-1974--Interface-Message-Processors-for-the-ARPA-Computer-Network--Report-2816--Quarterly-Technical-Report-5.pdf>)).
 16. ^ ^a ^b Bärwolff, M. (2010). "End-to-End Arguments in the Internet: Principles, Practices, and Theory". Self-published online and via Createspace/Amazon (PDF, errata, etc. (<http://xn--brwolff-5wa.de/publications/2010-10-PhD-thesis.html>))
 17. ^ Walden, D. C. (1974) *Some Changes to the IMP and the IMP/Host Interface*. RFC 660. Historic. NWG.
 18. ^ BBN (1974). *Interface Message Processors for the Arpa Computer Network*. BBN Report 2913. Quarterly Technical Report No. 7, 1 July 1974 to 30 September 1974. Bolt, Beranek and Newman Inc. (BBN).

External links

- MIT homepage of Jerome H. Saltzer (<http://web.mit.edu/Saltzer/>) featuring publication list, working papers, biography, etc.

- Personal homepage of David P. Reed (<http://reed.com/>) featuring publication list, blog, biography, etc.
- MIT homepage of David D. Clark (<http://groups.csail.mit.edu/ana/People/Clark.html>) featuring publication list, working papers, biography, etc.

Retrieved from "http://en.wikipedia.org/w/index.php?title=End-to-end_principle&oldid=480321632"

Categories: [Internet architecture](#) | [Network architecture](#) | [TCP/IP](#) | [Programming paradigms](#)

- This page was last modified on 5 March 2012 at 14:01.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

END-TO-END ARGUMENTS IN SYSTEM DESIGN

J.H. Saltzer, D.P. Reed and D.D. Clark*

M.I.T. Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgement. Low level mechanisms to support these functions are justified only as performance enhancements.

Introduction

Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer. Design principles that provide guidance in this choice of function placement are among the most important tools of a system designer. This paper discusses one class of function placement argument that has been used for many years with neither explicit recognition nor much conviction. However, the emergence of the data communication network as a computer system component has sharpened this line of function placement argument by making more apparent the situations in which and reasons why it applies. This paper articulates the argument explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements, and provides a rationale for moving function upward in a layered system, closer to the application that uses the function. We begin by considering the communication network version of the argument.

In a system that includes communications, one usually draws a modular boundary around the communication subsystem and defines a firm interface between it and the rest of the system. When doing so, it becomes apparent that there is a list of functions each of which might be implemented in any of several ways: by the communication subsystem, by its client, as a joint

* Authors' addresses: J.H. Saltzer and D.D. Clark, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139.; D.P. Reed, Software Arts, Inc., 27 Mica Lane, Wellesley, Massachusetts 02181.

This research was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense and monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Revised version of a paper from the Second International Conference on Distributed Computing Systems, Paris, France, April 8-10, 1981, pp. 509-512.: Copyright 1981 by The Institute of Electrical and Electronics Engineers, Inc. Reprinted with permission.

Published in ACM Transactions in Computer Systems 2, 4, November, 1984, pages 277-288.

Reprinted in Craig Partridge, editor Innovations in internetworking. Artech House, Norwood, MA, 1988, pages 195-206. ISBN 0-89006-337-0. Also scheduled to be reprinted in Amit Bhargava, editor. Integrated broadband networks. Artech House, Boston, 1991. ISBN 0-89006-483-0.

Scribe/FinalWord source: <http://web.mit.edu/Saltzer/www/publications/>

venture, or perhaps redundantly, each doing its own version. In reasoning about this choice, the requirements of the application provide the basis for a class of arguments, which go as follows:

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

We call this line of reasoning against low-level function implementation the "end-to-end argument." The following sections examine the end-to-end argument in detail, first with a case study of a typical example in which it is used – the function in question is reliable data transmission – and then by exhibiting the range of functions to which the same argument can be applied. For the case of the data communication system, this range includes encryption, duplicate message detection, message sequencing, guaranteed message delivery, detecting host crashes, and delivery receipts. In a broader context the argument seems to apply to many other functions of a computer operating system, including its file system. Examination of this broader context will be easier if we first consider the more specific data communication context, however.

End-to-end caretaking

Consider the problem of "careful file transfer." A file is stored by a file system, in the disk storage of computer A. Computer A is linked by a data communication network with computer B, which also has a file system and a disk store. The object is to move the file from computer A's storage to computer B's storage without damage, in the face of knowledge that failures can occur at various points along the way. The application program in this case is the file transfer program, part of which runs at host A and part at host B. In order to discuss the possible threats to the file's integrity in this transaction, let us assume that the following specific steps are involved:

1. At host A the file transfer program calls upon the file system to read the file from the disk, where it resides on several tracks, and the file system passes it to the file transfer program in fixed-size blocks chosen to be disk-format independent.
2. Also at host A the file transfer program asks the data communication system to transmit the file using some communication protocol that involves splitting the data into packets. The packet size is typically different from the file block size and the disk track size.
3. The data communication network moves the packets from computer A to computer B.
4. At host B a data communication program removes the packets from the data communication protocol and hands the contained data on to a second part of the file transfer application, the part that operates within host B.
5. At host B, the file transfer program asks the file system to write the received data on the disk of host B.

With this model of the steps involved, the following are some of the threats to the transaction that a careful designer might be concerned about:

1. The file, though originally written correctly onto the disk at host A, if read now may contain incorrect data, perhaps because of hardware faults in the disk storage system.
2. The software of the file system, the file transfer program, or the data communication system might make a mistake in buffering and copying the data of the file, either at host A or host B.
3. The hardware processor or its local memory might have a transient error while doing the buffering and copying, either at host A or host B.
4. The communication system might drop or change the bits in a packet, or lose a packet or deliver a packet more than once.

my thoughts

↓

Only the app

can do it -

so only the


app should

do it

Network should
not get involved

possible
errors

5. Either of the hosts may crash part way through the transaction after performing an unknown amount (perhaps all) of the transaction.

How would a careful file transfer application then cope with this list of threats? One approach might be to reinforce each of the steps along the way using duplicate copies, timeout and retry, carefully located redundancy for error detection, crash recovery, etc. The goal would be to reduce the probability of each of the individual threats to an acceptably small value. Unfortunately, systematic countering of threat two requires writing correct programs, which task is quite difficult, and not all the programs that must be correct are written by the file transfer application programmer. If we assume further that all these threats are relatively low in probability – low enough that the system allows useful work to be accomplished – brute force countermeasures such as doing everything three times appear uneconomical. 

The alternate approach might be called "end-to-end check and retry". Suppose that as an aid to coping with threat number one, stored with each file is a checksum that has sufficient redundancy to reduce the chance of an undetected error in the file to an acceptably negligible value. The application program follows the simple steps above in transferring the file from A to B. Then, as a final additional step, the part of the file transfer application residing in host B reads the transferred file copy back from its disk storage system into its own memory, recalculates the checksum, and sends this value back to host A, where it is compared with the checksum of the original. Only if the two checksums agree does the file transfer application declare the transaction committed. If the comparison fails, something went wrong, and a retry from the beginning might be attempted.

If failures really are fairly rare, this technique will normally work on the first try; occasionally a second or even third try might be required; one would probably consider two or more failures on the same file transfer attempt as indicating that some part of the system is in need of repair.

Now let us consider the usefulness of a common proposal, namely that the communication system provide, internally, a guarantee of reliable data transmission. It might accomplish this guarantee by providing selective redundancy in the form of packet checksums, sequence number checking, and internal retry mechanisms, for example. With sufficient care, the probability of undetected bit errors can be reduced to any desirable level. The question is whether or not this attempt to be helpful on the part of the communication system is useful to the careful file transfer application.

The answer is that threat number four may have been eliminated, but the careful file transfer application must still counter the remaining threats, so it should still provide its own retries based on an end-to-end checksum of the file. And if it does so, the extra effort expended in the communication system to provide a guarantee of reliable data transmission is only reducing the frequency of retries by the file transfer application; it has no effect on inevitability or correctness of the outcome, since correct file transmission is assured by the end-to-end checksum and retry whether or not the data transmission system is especially reliable.

Thus the argument: in order to achieve careful file transfer, the application program that performs the transfer must supply a file-transfer-specific, end-to-end reliability guarantee – in this case, a checksum to detect failures and a retry/commit plan. For the data communication system to go out of its way to be extraordinarily reliable does not reduce the burden on the application program to ensure reliability.

A too-real example

An interesting example of the pitfalls that one can encounter turned up recently at M.I.T.: One network system involving several local networks connected by gateways used a packet checksum on each hop from one gateway to the next, on the assumption that the primary threat to correct communication was corruption of bits during transmission. Application programmers, aware of

This doesn't seem totally provable - more like a theory

this checksum, assumed that the network was providing reliable transmission, without realizing that the transmitted data was unprotected while stored in each gateway. One gateway computer developed a transient error in which while copying data from an input to an output buffer a byte pair was interchanged, with a frequency of about one such interchange in every million bytes passed. Over a period of time many of the source files of an operating system were repeatedly transferred through the defective gateway. Some of these source files were corrupted by byte exchanges, and their owners were forced to the ultimate end-to-end error check: manual comparison with and correction from old listings.

Performance aspects

It would be too simplistic to conclude that the lower levels should play no part in obtaining reliability, however. Consider a network that is somewhat unreliable, dropping one message of each hundred messages sent. The simple strategy outlined above, transmitting the file and then checking to see that the file arrived correctly, would perform more poorly as the length of the file increases. The probability that all packets of a file arrive correctly decreases exponentially with the file length, and thus the expected time to transmit the file grows exponentially with file length. Clearly, some effort at the lower levels to improve network reliability can have a significant effect on application performance. But the key idea here is that the lower levels need not provide "perfect" reliability.

Thus the amount of effort to put into reliability measures within the data communication system is seen to be an engineering tradeoff based on performance, rather than a requirement for correctness. Note that performance has several aspects here. If the communication system is too unreliable, the file transfer application performance will suffer because of frequent retries following failures of its end-to-end checksum. If the communication system is beefed up with internal reliability measures, those measures have a performance cost, too, in the form of bandwidth lost to redundant data and delay added by waiting for internal consistency checks to complete before delivering the data. There is little reason to push in this direction very far, when it is considered that the end-to-end check of the file transfer application must still be implemented no matter how reliable the communication system becomes. The "proper" tradeoff requires careful thought; for example one might start by designing the communication system to provide just the reliability that comes with little cost and engineering effort, and then evaluate the residual error level to insure that it is consistent with an acceptable retry frequency at the file transfer level. It is probably not important to strive for a negligible error rate at any point below the application level.

Using performance to justify placing functions in a low-level subsystem must be done carefully. Sometimes, by examining the problem thoroughly, the same or better performance enhancement can be achieved at the high level. Performing a function at a low level may be more efficient, if the function can be performed with a minimum perturbation of the machinery already included in the low-level subsystem, but just the opposite situation can occur - that is, performing the function at the lower level may cost more - for two reasons. First, since the lower level subsystem is common to many applications, those applications that do not need the function will pay for it anyway. Second, the low-level subsystem may not have as much information as the higher levels, so it cannot do the job as efficiently.

Frequently, the performance tradeoff is quite complex. Consider again the careful file transfer on an unreliable network. The usual technique for increasing packet reliability is some sort of per-packet error check with a retry protocol. This mechanism can be implemented either in the communication subsystem or in the careful file transfer application. For example, the receiver in the careful file transfer can periodically compute the checksum of the portion of the file thus far received and transmit this back to the sender. The sender can then restart by retransmitting any portion that arrived in error.

oops
instead ignore
and have
endpoint
do it

Can still have for performance?

actually @ low level often seems to cost more

The end-to-end argument does not tell us where to put the early checks, since either layer can do this performance-enhancement job. Placing the early retry protocol in the file transfer application simplifies the communication system, but may increase overall cost, since the communication system is shared by other applications and each application must now provide its own reliability enhancement. Placing the early retry protocol in the communication system may be more efficient, since it may be performed inside the network on a hop-by-hop basis, reducing the delay involved in correcting a failure. At the same time, there may be some application that finds the cost of the enhancement is not worth the result but it now has no choice in the matter*. A great deal of information about system implementation is needed to make this choice intelligently.

Other examples of the end-to-end argument

just let app decide

Delivery guarantees

The basic argument that a lower-level subsystem that supports a distributed application may be wasting its effort providing a function that must by nature be implemented at the application level anyway can be applied to a variety of functions in addition to reliable data transmission. Perhaps the oldest and most widely known form of the argument concerns acknowledgement of delivery. A data communication network can easily return an acknowledgement to the sender for every message delivered to a recipient. The ARPANET, for example, returns a packet known as "Request For Next Message" (RFNM)[1] whenever it delivers a message. Although this acknowledgement may be useful within the network as a form of congestion control (originally the ARPANET refused to accept another message to the same target until the previous RFNM had returned) it was never found to be very helpful to applications using the ARPANET. The reason is that knowing for sure that the message was delivered to the target host is not very important. What the application wants to know is whether or not the target host acted on the message; all manner of disaster might have struck after message delivery but before completion of the action requested by the message. The acknowledgement that is really desired is an end-to-end one, which can be originated only by the target application – "I did it", or "I didn't."

app is the only one who can confirm

Another strategy for obtaining immediate acknowledgements is to make the target host sophisticated enough that when it accepts delivery of a message it also accepts responsibility for guaranteeing that the message is acted upon by the target application. This approach can eliminate the need for an end-to-end acknowledgement in some, but not all applications. An end-to-end acknowledgement is still required for applications in which the action requested of the target host should be done only if similar actions requested of other hosts are successful. This kind of application requires a two-phase commit protocol[5,10,15], which is a sophisticated end-to-end acknowledgement. Also, if the target application may either fail or refuse to do the requested action, and thus a negative acknowledgement is a possible outcome, an end-to-end acknowledgement may still be a requirement.

Secure transmission of data

Another area in which an end-to-end argument can be applied is that of data encryption. The argument here is threefold. First, if the data transmission system performs encryption and decryption, it must be trusted to manage securely the required encryption keys. Second, the data will be in the clear and thus vulnerable as it passes into the target node and is fanned out to the target application. Third, the authenticity of the message must still be checked by the application. If the application performs end-to-end encryption, it obtains its required authentication check, it

inside the computer from the network stack

* For example, real time transmission of speech has tighter constraints on message delay than on bit-error rate. Most retry schemes significantly increase the variability of delay.

layering

can handle key management to its satisfaction, and the data is never exposed outside the application.

Thus, to satisfy the requirements of the application, there is no need for the communication subsystem to provide for automatic encryption of all traffic. Automatic encryption of all traffic by the communication subsystem may be called for, however, to ensure something else – that a misbehaving user or application program does not deliberately transmit information that should not be exposed. The automatic encryption of all data as it is put into the network is one more firewall the system designer can use to ensure that information does not escape outside the system. Note however, that this is a different requirement from authenticating access rights of a system user to specific parts of the data. This network-level encryption can be quite unsophisticated – the same key can be used by all hosts, with frequent changes of the key. No per-user keys complicate the key management problem. The use of encryption for application-level authentication and protection is complementary. Neither mechanism can satisfy both requirements completely.

Duplicate message suppression

A more sophisticated argument can be applied to duplicate message suppression. A property of some communication network designs is that a message or a part of a message may be delivered twice, typically as a result of time-out-triggered failure detection and retry mechanisms operating within the network. The network can provide the function of watching for and suppressing any such duplicate messages, or it can simply deliver them. One might expect that an application would find it very troublesome to cope with a network that may deliver the same message twice; indeed it is troublesome. Unfortunately, even if the network suppresses duplicates, the application itself may accidentally originate duplicate requests, in its own failure/retry procedures. These application level duplications look like different messages to the communication system, so it cannot suppress them; suppression must be accomplished by the application itself with knowledge of how to detect its own duplicates.

A common example of duplicate suppression that must be handled at a high level is when a remote system user, puzzled by lack of response, initiates a new login to a time-sharing system. For another example, most communication applications involve a provision for coping with a system crash at one end of a multi-site transaction: reestablish the transaction when the crashed system comes up again. Unfortunately, reliable detection of a system crash is problematical: the problem may just be a lost or long-delayed acknowledgement. If so, the retried request is now a duplicate, which only the application can discover. Thus the end-to-end argument again: if the application level has to have a duplicate-suppressing mechanism anyway, that mechanism can also suppress any duplicates generated inside the communication network, so the function can be omitted from that lower level. The same basic reasoning applies to completely omitted messages as well as to duplicated ones.

Guaranteeing FIFO message delivery

Ensuring that messages arrive at the receiver in the same order they are sent is another function usually assigned to the communication subsystem. The mechanism usually used to achieve such first-in, first-out (FIFO) behavior guarantees FIFO ordering among messages sent on the same virtual circuit. Messages sent along independent virtual circuits, or through intermediate processes outside the communication subsystem may arrive in an order different from the order sent. A distributed application in which one node can originate requests that initiate actions at several sites cannot take advantage of the FIFO ordering property to guarantee that the actions requested occur in the correct order. Instead, an independent mechanism at a higher level than the communication subsystem must control the ordering of actions.

App should provide itself

read so
can answer
quiz qu

like a VPN

Transaction management

We have now applied the end-to-end argument in the construction of the SWALLOW distributed data storage system[15], where it leads to significant reduction in overhead. SWALLOW provides data storage servers called repositories that can be used remotely to store and retrieve data. Accessing data at a repository is done by sending it a message specifying the object to be accessed, the version, and type of access (read/write), plus a value to be written if the access is a write. The underlying message communication system does not suppress duplicate messages, since a) the object identifier plus the version information suffices to detect duplicate writes, and b) the effect of a duplicate read request message is only to generate a duplicate response, which is easily discarded by the originator. Consequently, the low-level message communication protocol is significantly simplified.

The underlying message communication system does not provide delivery acknowledgement either. The acknowledgement that the originator of a write request needs is that the data was stored safely. This acknowledgement can be provided only by high levels of the SWALLOW system. For read requests, a delivery acknowledgement is redundant, since the response containing the value read is sufficient acknowledgement. By eliminating delivery acknowledgements, the number of messages transmitted is halved. This message reduction can have a significant effect on both host load and network load, improving performance. This same line of reasoning has also been used in development of an experimental protocol for remote access to disk records[6]. The resulting reduction in path length in lower-level protocols was important in maintaining good performance on remote disk access.

Identifying the ends

do what's best for the app

Using the end-to-end argument sometimes requires subtlety of analysis of application requirements. For example, consider a computer communication network that carries some packet voice connections, conversations between digital telephone instruments. For those connections that carry voice packets, an unusually strong version of the end-to-end argument applies: if low levels of the communication system try to accomplish bit-perfect communication, they will probably introduce uncontrolled delays in packet delivery, for example, by requesting retransmission of damaged packets and holding up delivery of later packets until earlier ones have been correctly retransmitted. Such delays are disruptive to the voice application, which needs to feed data at a constant rate to the listener. It is better to accept slightly damaged packets as they are, or even to replace them with silence, a duplicate of the previous packet, or a noise burst. The natural redundancy of voice, together with the high-level error correction procedure in which one participant says "excuse me, someone dropped a glass. Would you please say that again?" will handle such dropouts, if they are relatively infrequent.

The humans retry! - not the computer

However, this strong version of the end-to-end argument is a property of the specific application – two people in real-time conversation – rather than a property, say, of speech in general. If one considers instead a speech message system, in which the voice packets are stored in a file for later listening by the recipient, the arguments suddenly change their nature. Short delays in delivery of packets to the storage medium are not particularly disruptive so there is no longer any objection to low-level reliability measures that might introduce delay in order to achieve reliability. More important, it is actually helpful to this application to get as much accuracy as possible in the recorded message, since the recipient, at the time of listening to the recording, is not going to be able to ask the sender to repeat a sentence. On the other hand, with a storage system acting as the receiving end of the voice communication, an end-to-end argument does apply to packet ordering and duplicate suppression. Thus the end-to-end argument is not an absolute rule, but rather a guideline that helps in application and protocol design analysis; one must use some care to identify the end points to which the argument should be applied.

hard to ask

well the app could decide

History, and application to other system areas

The individual examples of end-to-end arguments cited in this paper are not original; they have accumulated over the years. The first example of questionable intermediate delivery acknowledgements noticed by the authors was the "wait" message of the M.I.T. Compatible Time-Sharing System, which the system printed on the user's terminal whenever the user entered a command[3]. (The message had some value in the early days of the system, when crashes and communication failures were so frequent that intermediate acknowledgements provided some needed reassurance that all was well.) *hahq*

The end-to-end argument relating to encryption was first publicly discussed by Branstad in a 1973 paper[2]; presumably the military security community held classified discussions before that time. Diffie and Hellman[4] and Kent[8] develop the arguments in more depth, and Needham and Schroeder[11] devised improved protocols for the purpose.

The two-phase-commit data update protocols of Gray[5], Lampson and Sturgis[10] and Reed[13] all use a form of end-to-end argument to justify their existence; they are end-to-end protocols that do not depend for correctness on reliability, FIFO sequencing, or duplicate suppression within the communication system, since all of these problems may also be introduced by other system component failures as well. Reed makes this argument explicitly in the second chapter of his Ph.D. thesis on decentralized atomic actions[14].

End-to-end arguments are often applied to error control and correctness in application systems. For example, a banking system usually provides high-level auditing procedures as a matter of policy and legal requirement. Those high-level auditing procedures will uncover not only high-level mistakes such as performing a withdrawal against the wrong account, it will also detect low-level mistakes such as coordination errors in the underlying data management system. Therefore a costly algorithm that absolutely eliminates such coordination errors may be arguably less appropriate than a less costly algorithm that just makes such errors very rare. In airline reservation systems, an agent can be relied upon to keep trying, through system crashes and delays, until a reservation is either confirmed or refused. Lower level recovery procedures to guarantee that an unconfirmed request for a reservation will survive a system crash are thus not vital. In telephone exchanges, a failure that could cause a single call to be lost is considered not worth providing explicit recovery for, since the caller will probably replace the call if it matters[7]: All of these design approaches are examples of the end-to-end argument being applied to automatic recovery.

Much of the debate in the network protocol community over datagrams, virtual circuits, and connectionless protocols is a debate about end-to-end arguments. A modularity argument prizes a reliable, FIFO sequenced, duplicate-suppressed stream of data as a system component that is easy to build on, and that argument favors virtual circuits. The end-to-end argument claims that centrally-provided versions of each of those functions will be incomplete for some applications, and those applications will find it easier to build their own version of the functions starting with datagrams.

A version of the end-to-end argument in a non-communication application was developed in the 1950's by system analysts whose responsibility included reading and writing files on large numbers of magnetic tape reels. Repeated attempts to define and implement a "reliable tape subsystem" repeatedly foundered, as flaky tape drives, undependable system operators, and system crashes conspired against all narrowly focused reliability measures. Eventually, it became standard practice for every application to provide its own application-dependent checks and recovery strategy; and to assume that lower-level error detection mechanisms at best reduced the frequency with which the higher-level checks failed. As an example, the Multics file backup system[17], even though it is built on a foundation of a magnetic tape subsystem format that

provides very powerful error detection and correction features, provides its own error control in the form of record labels and multiple copies of every file.

The arguments that are used in support of reduced instruction set computer (RISC) architecture are similar to end-to-end arguments. The RISC argument is that the client of the architecture will get better performance by implementing exactly the instructions needed from primitive tools; any attempt by the computer designer to anticipate the client's requirements for an esoteric feature will probably miss the target slightly and the client will end up reimplementing that feature anyway. (We are indebted to M. Satyanarayanan for pointing out this example.)

Lampson, in his arguments supporting the "open operating system,"[9] uses an argument similar to the end-to-end argument as a justification. Lampson argues against making any function a permanent fixture of lower-level modules; the function may be provided by a lower-level module but it should always be replaceable by an application's special version of the function. The reasoning is that for any function you can think of, at least some applications will find that by necessity they must implement the function themselves in order to meet correctly their own requirements. This line of reasoning leads Lampson to propose an "open" system in which the entire operating system consists of replaceable routines from a library. Such an approach has only recently become feasible in the context of computers dedicated to a single application. It may be the case that the large quantity of fixed supervisor function typical of large-scale operating systems is only an artifact of economic pressures that demanded multiplexing of expensive hardware and therefore a protected supervisor. Most recent system "kernelization" projects, in fact, have focused at least in part on getting function out of low system levels[16,12]. Though this function movement is inspired by a different kind of correctness argument, it has the side effect of producing an operating system that is more flexible for applications, which is exactly the main thrust of the end-to-end argument.

Conclusions

End-to-end arguments are a kind of "Occam's razor" when it comes to choosing the functions to be provided in a communication subsystem. Because the communication subsystem is frequently specified before applications that use the subsystem are known, the designer may be tempted to "help" the users by taking on more function than necessary. Awareness of end-to-end arguments can help to reduce such temptations.

It is fashionable these days to talk about "layered" communication protocols, but without clearly defined criteria for assigning functions to layers. Such layerings are desirable to enhance modularity. End-to-end arguments may be viewed as part of a set of rational principles for organizing such layered systems. We hope that our discussion will help to add substance to arguments about the "proper" layering.

Acknowledgements

Many people have read and commented on an earlier draft of this paper, including David Cheriton, F.B. Schneider, and Liba Svobodova. The subject was also discussed at the ACM Workshop in Fundamentals of Distributed Computing, in Fallbrook, California during December 1980. Those comments and discussions were quite helpful in clarifying the arguments.

References

1. Bolt Beranek and Newman Inc. Specifications for the interconnection of a host and an IMP. Technical Report No. 1822, Cambridge, Mass., December, 1981.
2. Branstad, D.K. Security aspects of computer networks. AIAA Paper No. 73-427, AIAA Computer Network Systems Conference, Huntsville, Alabama, April, 1973.
3. Corbato, F.J., et al. *The Compatible Time-Sharing System, A Programmer's Guide*. M.I.T. Press, Cambridge, Massachusetts, 1963, p.10.
4. Diffie, W., and Hellman, M.E. New directions in cryptography. *IEEE Trans. on Info. Theory*, IT-22, 6, (November, 1976), pp.644-654.
5. Gray, J.N. Notes on database operating systems. In *Operating System: An Advanced Course*. Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978, pp.393-481.
6. Greenwald, M. Remote virtual disk protocol specifications. M.I.T. Laboratory for Computer Science Technical Memorandum, in preparation. Expected publication, 1984.
7. Keister, W., Ketchledge, R.W., and Vaughan, H.E.: No. 1 ESS: System organization and objectives. *Bell System Technical Journal* 53, 5 (part 1), (September, 1964) p. 1841.
8. Kent, S.T.: Encryption-based protection protocols for interactive user-computer communication.: S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May, 1976. Also available as M.I.T. Laboratory for Computer Science Technical Report, TR-162, May, 1976.
9. Lampson, B.W., and Sproull, R.F. An open operating system for a single-user machine. *Proc. Seventh Symposium on Operating Systems Principles, Operating Systems Review* 13, Special issue (December, 1979), pp.98-105.
10. Lampson, B., and Sturgis, H: Crash recovery in a distributed data storage system. Working paper, Xerox PARC, November, 1976 and April, 1979. Submitted to *CACM*.
11. Needham, R.M., and Schroeder, M.D.: Using encryption for authentication in large networks of computers. *CACM* 21, 12, (December, 1978), pp.993-999.
12. Popek, G.J., et al.: UCLA data secure unix. *Proc. 1979 NCC*, AFIPS Press, pp.355-364.
13. Reed, D.P.: Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems* 1, 1 (February, 1983), pp.3-23.
14. Reed, D.P.: Naming and synchronization in a decentralized computer system. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1978. Also available as M.I.T. Laboratory for Computer Science Technical Report, TR-205, September, 1978.
15. Reed, D.P., and Svobodova, L.: SWALLOW: A distributed data storage system for a local network. In West, A., and Janson, P., ed. *Local Networks for Computer Communications, Proc. IFIP Working Group 6.4 International Workshop on Local Networks*. North-Holland, Amsterdam, 1981, pp.355-373.
16. Schroeder, M.D., Clark, D.D., and Saltzer, J.H.: The Multics kernel design project. *Proc. Sixth Symposium on Operating Systems Principles, Operating Systems Review* 11, 5 (November, 1977,) pp.43-56.
17. Stern, J.A.: Backup and recovery of on-line information in a computer utility. S.M. thesis, M.I.T. Department of Electrical Engineering and Computer Science, August 1973. Available as M.I.T. Project MAC Technical Report TR-116, January, 1974.

I know a lot about this...

Network address translation

From Wikipedia, the free encyclopedia

breaks end-to-end

But allows for more features

In computer networking, **network address translation (NAT)** is the process of modifying IP address information in IP packet headers while in transit across a traffic routing device.

The simplest type of NAT provides a one to one translation of IP addresses. RFC 2663 refers to this type of NAT as **basic NAT**. It is often also referred to as **one-to-one NAT**. In this type of NAT only the IP addresses, IP header checksum and any higher level checksums that include the IP address need to be changed. The rest of the packet can be left untouched (at least for basic TCP/UDP functionality, some higher level protocols may need further translation). Basic NATs can be used when there is a requirement to interconnect two IP networks with incompatible addressing.

most common

However it is common to hide an entire IP address space, usually consisting of private IP addresses, behind a single IP address (or in some cases a small group of IP addresses) in another (usually public) address space. To avoid ambiguity in the handling of returned packets, a one-to-many NAT must alter higher level information such as TCP/UDP ports in outgoing communications and must maintain a translation table so that return packets can be correctly translated back. RFC 2663 uses the term **NAPT (network address and port translation)** for this type of NAT. Other names include **PAT (port address translation)**, **IP masquerading**, **NAT Overload** and **many-to-one NAT**. Since this is the most common type of NAT it is often referred to simply as NAT.

As described, the method enables communication through the router only when the conversation originates in the masqueraded network, since this establishes the translation tables. For example, a web browser in the masqueraded network can browse a website outside, but a web browser outside could not browse a web site in the masqueraded network. However, most NAT devices today allow the network administrator to configure translation table entries for permanent use. This feature is often referred to as "static NAT" or port forwarding and allows traffic originating in the "outside" network to reach designated hosts in the masqueraded network.

Security

Port IP forwarding

In the mid-1990s NAT became a popular tool for alleviating the consequences of IPv4 address exhaustion.^[1] It has become a common, indispensable feature in routers for home and small-office Internet connections. Most systems using NAT do so in order to enable multiple hosts on a private network to access the Internet using a single public IP address.

Network address translation has serious drawbacks on the quality of Internet connectivity and requires careful attention to the details of its implementation. In particular all types of NAT break the originally envisioned model of IP end-to-end connectivity across the Internet and NAPT makes it difficult for systems behind a NAT to accept incoming communications. As a result, NAT traversal methods have been devised to alleviate the issues encountered.

security

like Skype

Contents

- 1 One to many NATs
 - 1.1 Methods of Port translation

- 1.2 Type of NAT and NAT Traversal
- 2 Implementation
 - 2.1 Establishing Two-Way Communication
 - 2.2 An Analogy
 - 2.3 Translation of the Endpoint
 - 2.4 Visibility of Operation
- 3 NAT and TCP/UDP
- 4 Destination network address translation (DNAT)
- 5 SNAT
 - 5.1 Secure network address translation
- 6 Dynamic network address translation
- 7 Applications affected by NAT
- 8 Advantages of PAT
- 9 Drawbacks
- 10 Specifications
- 11 Examples of NAT software
- 12 See also
- 13 References
- 14 External links

One to many NATs

The majority of NATs map multiple private hosts to one publicly exposed IP address. In a typical configuration, a local network uses one of the designated "private" IP address subnets (RFC 1918). A router on that network has a private address in that address space. The router is also connected to the Internet with a "public" address assigned by an Internet service provider. As traffic passes from the local network to the Internet, the source address in each packet is translated on the fly from a private address to the public address. The router tracks basic data about each active connection (particularly the destination address and port). When a reply returns to the router, it uses the connection tracking data it stored during the outbound phase to determine the private address on the internal network to which to forward the reply.

All Internet packets have a source IP address and a destination IP address. Typically packets passing from the private network to the public network will have their source address modified while packets passing from the public network back to the private network will have their destination address modified. More complex configurations are also possible.

To avoid ambiguity in how to translate returned packets, further modifications to the packets are required. The vast bulk of Internet traffic is TCP and UDP packets and for these protocols the port numbers are changed so that the combination of IP and port information on the returned packet can be unambiguously mapped to the corresponding private address and port information. Protocols not based on TCP or UDP require other translation techniques. ICMP packets typically relate to an existing connection and need to be mapped using the same IP and port mappings as that connection.

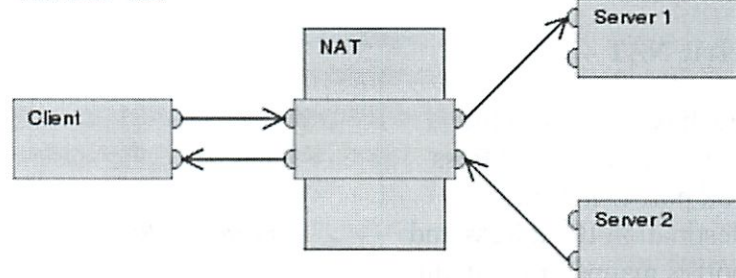
Methods of Port translation

There are several ways of implementing network address and port translation. In some application protocols that use IP address information, the application running on a node in the masqueraded network needs to determine the external address of the NAT, i.e., the address that its communication peers detect, and, furthermore, often needs to examine and categorize the type of mapping in use. Usually this is done because it is desired to set up a direct communications path (either to save the cost of taking the data via a server or to improve performance) between two clients both of which are behind separate NATs. For this purpose, the Simple traversal of UDP over NATs (STUN) protocol was developed (RFC 3489, March 2003). It classified NAT implementation as *full cone NAT*, *(address) restricted cone NAT*, *port restricted cone NAT* or *symmetric NAT* and proposed a methodology for testing a device accordingly. However, these procedures have since been deprecated from standards status, as the methods have proven faulty and inadequate to correctly assess many devices. New methods have been standardized in RFC 5389 (October 2008) and the STUN acronym now represents the new title of the specification: *Session Traversal Utilities for NAT*.

Full-cone NAT, also known as *one-to-one NAT*

- Once an internal address (iAddr:iPort) is mapped to an external address (eAddr:ePort), any packets from iAddr:iPort will be sent through eAddr:ePort.
- Any external host can send packets to iAddr:iPort by sending packets to eAddr:ePort.

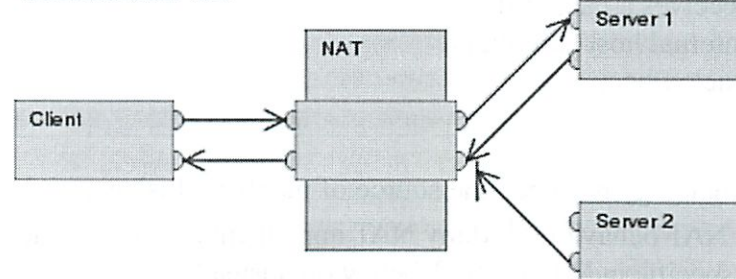
"Full Cone" NAT



(Address) restricted cone NAT

- Once an internal address (iAddr:iPort) is mapped to an external address (eAddr:ePort), any packets from iAddr:iPort will be sent through eAddr:ePort.
- An external host (hAddr:any) can send packets to iAddr:iPort by sending packets to eAddr:ePort only if iAddr:iPort has previously sent a packet to hAddr:any. "Any" means the port number doesn't matter.

"Restricted Cone" NAT



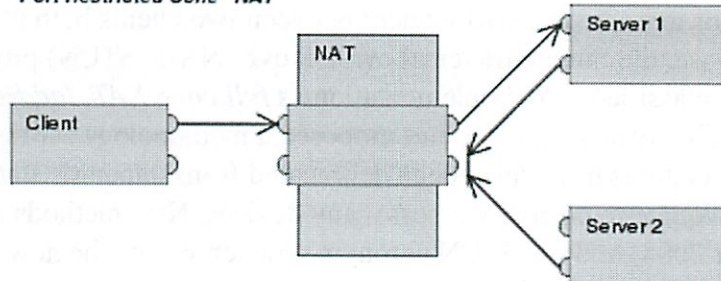
Port-restricted cone NAT

Like an address restricted cone

NAT, but the restriction includes port numbers.

- Once an internal address (*iAddr:iPort*) is mapped to an external address (*eAddr:ePort*), any packets from *iAddr:iPort* will be sent through *eAddr:ePort*.
- An external host (*hAddr:hPort*) can send packets to *iAddr:iPort* by sending packets to *eAddr:ePort* only if *iAddr:iPort* has previously sent a packet to *hAddr:hPort*.

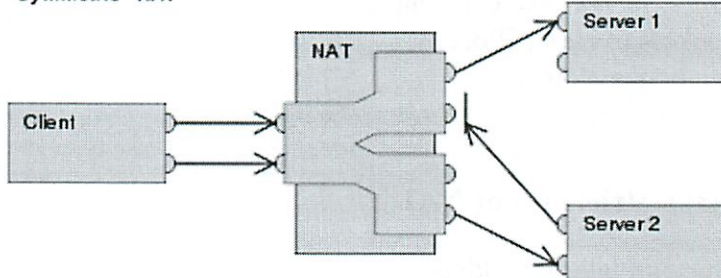
"Port Restricted Cone" NAT



Symmetric NAT

- Each request from the same internal IP address and port to a specific destination IP address and port is mapped to a unique external source IP address and port, if the same internal host sends a packet even with the same source address and port but to a different destination, a different mapping is used.
- Only an external host that receives a packet from an internal host can send a packet back.

"Symmetric" NAT



This terminology has been the source of much confusion, as it has proven inadequate at describing real-life NAT behavior.^[2] Many NAT implementations combine these types, and it is therefore better to refer to specific individual NAT behaviors instead of using the Cone/Symmetric terminology. Especially, most NAT translators combine *symmetric NAT* for outgoing connections with *static port mapping*, where incoming packets to the external address and port are redirected to a specific internal address and port. Some products can redirect packets to several internal hosts, e.g. to divide the load between a few servers. However, this introduces problems with more sophisticated communications that have many interconnected packets, and thus is rarely used.

Type of NAT and NAT Traversal

Sype, BitTorrent

The NAT traversal problem arises when two peers behind distinct NAT try to communicate. One way to solve this problem is to use port forwarding, another way is to use various NAT traversal techniques. The most popular technique for TCP NAT traversal is TCP hole punching, which requires the NAT to follow the port preservation design for TCP, as explained below.

Many NAT implementations follow the *port preservation* design especially for TCP, which is to say that they use the same values as internal and external port numbers. NAT *port preservation* for outgoing TCP connections is especially important for TCP NAT traversal, because programs usually bind distinct TCP sockets to ephemeral ports for distinct TCP connections, rendering NAT port prediction impossible for TCP.

On the other hand, for UDP, NATs do not need to have *port preservation* because applications usually reuse the same UDP socket to send packets to distinct hosts, making port prediction straightforward, as it is the same source port for each packet.

Furthermore, *port preservation* in NAT for TCP allows P2P protocols to offer less complexity and less latency because there is no need to use a third party to discover the NAT port since the application already knows the NAT port.^[3]

However, if two internal hosts attempt to communicate with the same external host using the same port number, the external port number used by the second host will be chosen at random. Such NAT will be sometimes perceived as (*address*) *restricted cone NAT* and other times as *symmetric NAT*.

Recent studies have shown that roughly 70% of clients in P2P networks employ some form of NAT.^[4]

Implementation

Establishing Two-Way Communication

Every TCP and UDP packet contains both a source IP address and source port number as well as a destination IP address and destination port number. The port address/IP address pair forms a socket. In particular, the source port address and source IP address form the source socket.

For publicly accessible services such as web servers and mail servers the port number is important. For example, port 80 connects to the web server software and port 25 to a mail server's SMTP daemon. The IP address of a public server is also important, similar in global uniqueness to a postal address or telephone number. Both IP address and port must be correctly known by all hosts wishing to successfully communicate.

Private IP addresses as described in RFC 1918 are significant only on private networks where they are used, which is also true for host ports. Ports are unique endpoints of communication on a host, so a connection through the NAT device is maintained by the combined mapping of port and IP address.

PAT resolves conflicts that would arise through two different hosts using the same source port number to establish unique connections at the same time.

An Analogy

A NAT device is similar to a phone system at an office that has one public telephone number and multiple extensions. Outbound phone calls made from the office all appear to come from the same telephone number. However, an incoming call that does not specify an extension cannot be transferred to an individual inside the office. In this scenario, the office is a private LAN, the main phone number is the public IP address, and the individual extensions are unique port numbers.^[5]

Translation of the Endpoint

With NAT, all communication sent to external hosts actually contain the *external* IP address and port information of the NAT device instead of internal host IPs or port numbers.

- When a computer on the private (internal) network sends a packet to the external network, the NAT device replaces the internal IP address in the source field of the packet header (*sender's address*) with the external IP address of the NAT device. NAT may then assign the connection a port number from a pool of available ports, inserting this port number in the source port field (much like the *post office box number*), and forwards the packet to the external network. The NAT device then makes an entry in a translation table containing the internal IP address, original source port, and the translated source port. Subsequent packets from the same connection are translated to the same port number.
- The computer receiving a packet that has undergone NAT establishes a connection to the port and IP address specified in the altered packet, oblivious to the fact that the supplied address is being translated (analogous to using a *post office box number*).
- A packet coming from the external network is mapped to a corresponding internal IP address and port number from the translation table, replacing the external IP address and port number in the incoming packet header (similar to the translation from *post office box number* to *street address*). The packet is then forwarded over the inside network. Otherwise, if the destination port number of the incoming packet is not found in the translation table, the packet is dropped or rejected because the NAT device doesn't know where to send it.

NAT will only translate IP addresses and ports of its internal hosts, hiding the true endpoint of an internal host on a private network.

Visibility of Operation

NAT operation is typically transparent to both the internal and external hosts.

Typically the internal host is aware of the true IP address and TCP or UDP port of the external host. Typically the NAT device may function as the default gateway for the internal host. However the external host is only aware of the public IP address for the NAT device and the particular port being used to communicate on behalf of a specific internal host.

External server can know

NAT and TCP/UDP

"Pure NAT", operating on IP alone, may or may not correctly parse protocols that are totally concerned with IP information, such as ICMP, depending on whether the payload is interpreted by a host on the "inside" or "outside" of translation. As soon as the protocol stack is traversed, even with such basic

NAT provides a one-to-one internal to public static IP address mapping, dynamic NAT doesn't make the mapping to the public IP address static and usually uses a group of available public IP addresses.

Applications affected by NAT

Some Application Layer protocols (such as FTP and SIP) send explicit network addresses within their application data. FTP in active mode, for example, uses separate connections for control traffic (commands) and for data traffic (file contents). When requesting a file transfer, the host making the request identifies the corresponding data connection by its network layer and transport layer addresses. If the host making the request lies behind a simple NAT firewall, the translation of the IP address and/or TCP port number makes the information received by the server invalid. The Session Initiation Protocol (SIP) controls many Voice over IP (VoIP) calls, and suffers the same problem. SIP and SDP may use multiple ports to set up a connection and transmit voice stream via RTP. IP addresses and port numbers are encoded in the payload data and must be known prior to the traversal of NATs. Without special techniques, such as STUN, NAT behavior is unpredictable and communications may fail.

I must know about apps
Application layer gateway (ALG) software or hardware may correct these problems. An ALG software module running on a NAT firewall device updates any payload data made invalid by address translation. ALGs obviously need to understand the higher-layer protocol that they need to fix, and so each protocol with this problem requires a separate ALG. For example, on many Linux systems, there are kernel modules called *connection trackers* which serve to implement ALGs. However, ALG does not work if the control channel is encrypted (e.g. FTPS).

Another possible solution to this problem is to use NAT traversal techniques using protocols such as STUN or ICE, or proprietary approaches in a session border controller. NAT traversal is possible in both TCP- and UDP-based applications, but the UDP-based technique is simpler, more widely understood, and more compatible with legacy NATs.^[citation needed] In either case, the high level protocol must be designed with NAT traversal in mind, and it does not work reliably across symmetric NATs or other poorly-behaved legacy NATs.

Other possibilities are UPnP (Universal Plug and Play) or NAT-PMP (NAT Port Mapping Protocol), but these require the cooperation of the NAT device.

Most traditional client-server protocols (FTP being the main exception), however, do not send layer 3 contact information and therefore do not require any special treatment by NATs. In fact, avoiding NAT complications is practically a requirement when designing new higher-layer protocols today (e.g. the use of SFTP instead of FTP).

NATs can also cause problems where IPsec encryption is applied and in cases where multiple devices such as SIP phones are located behind a NAT. Phones which encrypt their signaling with IPsec encapsulate the port information within an encrypted packet, meaning that NA(P)T devices cannot access and translate the port. In these cases the NA(P)T devices revert to simple NAT operation. This means that all traffic returning to the NAT will be mapped onto one client causing service to more than one client "behind" the NAT to fail. There are a couple of solutions to this problem: one is to use TLS, which operates at level 4 in the OSI Reference Model and therefore does not mask the port number; another is to encapsulate the IPsec within UDP - the latter being the solution chosen by TISPAN to achieve secure NAT traversal.

hacks

protocols as TCP and UDP, the protocols will break unless NAT takes action beyond the network layer.

IP packets have a checksum in each packet header, which provides error detection only for the header. IP datagrams may become fragmented and it is necessary for a NAT to reassemble these fragments to allow correct recalculation of higher-level checksums and correct tracking of which packets belong to which connection.

The major transport layer protocols, TCP and UDP, have a checksum that covers all the data they carry, as well as the TCP/UDP header, plus a "pseudo-header" that contains the source and destination IP addresses of the packet carrying the TCP/UDP header. For an originating NAT to pass TCP or UDP successfully, it must recompute the TCP/UDP header checksum based on the translated IP addresses, not the original ones, and put that checksum into the TCP/UDP header of the first packet of the fragmented set of packets. The receiving NAT must recompute the IP checksum on every packet it passes to the destination host, and also recognize and recompute the TCP/UDP header using the retranslated addresses and pseudo-header. This is not a completely solved problem. One solution is for the receiving NAT to reassemble the entire segment and then recompute a checksum calculated across all packets.

The originating host may perform Maximum transmission unit (MTU) path discovery to determine the packet size that can be transmitted without fragmentation, and then set the *don't fragment* (DF) bit in the appropriate packet header field.

Destination network address translation (DNAT)

DNAT is a technique for transparently changing the destination IP address of an en-route packet and performing the inverse function for any replies. Any router situated between two endpoints can perform this transformation of the packet.

DNAT is commonly used to publish a service located in a private network on a publicly accessible IP address. This use of DNAT is also called port forwarding, or DMZ when used on an entire server, which becomes exposed to the WAN, becoming analogous to an undefended military demilitarised zone (DMZ).

SNAT

The meaning of the term *SNAT* varies by vendor. Many vendors have proprietary definitions for *SNAT*. A common expansion is *source NAT*, the counterpart of *destination NAT* (*DNAT*). Microsoft uses the acronym for *Secure NAT*, in regard to the ISA Server. For Cisco Systems, *SNAT* means *stateful NAT*.

Secure network address translation

In computer networking, the process of network address translation done in a secure way involves rewriting the source and/or destination addresses of IP packets as they pass through a router or firewall.

Dynamic network address translation

Dynamic NAT, just like static NAT, is not common in smaller networks but is found within larger corporations with complex networks. The way dynamic NAT differs from static NAT is that where static

The DNS protocol vulnerability announced by Dan Kaminsky on July 8, 2008 is indirectly affected by NAT port mapping. To avoid DNS server cache poisoning, it is highly desirable to not translate UDP source port numbers of outgoing DNS requests from a DNS server which is behind a firewall which implements NAT. The recommended work-around for the DNS vulnerability is to make all caching DNS servers use randomized UDP source ports. If the NAT function de-randomizes the UDP source ports, the DNS server will be made vulnerable.

Advantages of PAT

In addition to the advantages provided by NAT:

- PAT (Port Address Translation) allows many internal hosts to share a single external IP address.
- Users who do not require support for inbound connections do not consume public IP addresses.

Drawbacks

The primary purpose of IP-masquerading NAT is that it has been a practical solution to the impending exhaustion of IPv4 address space. Even large networks can be connected to the Internet with as little as a single IP address. The more common arrangement is having machines that require end-to-end connectivity supplied with a routable IP address, while having machines that do not provide services to outside users behind NAT with only a few IP addresses used to enable Internet access, however, this brings some problems, outlined below.

Some^[6] have also called this exact feature a major drawback, since it delays the need for the implementation of IPv6:

"[...] it is possible that its [NAT's] widespread use will significantly delay the need to deploy IPv6. [...] It is probably safe to say that networks would be better off without NAT [...]"

Hosts behind NAT-enabled routers do not have end-to-end connectivity and cannot participate in some Internet protocols. Services that require the initiation of TCP connections from the outside network, or stateless protocols such as those using UDP, can be disrupted. Unless the NAT router makes a specific effort to support such protocols, incoming packets cannot reach their destination. Some protocols can accommodate one instance of NAT between participating hosts ("passive mode" FTP, for example), sometimes with the assistance of an application-level gateway (see below), but fail when both systems are separated from the Internet by NAT. Use of NAT also complicates tunneling protocols such as IPsec because NAT modifies values in the headers which interfere with the integrity checks done by IPsec and other tunneling protocols.

End-to-end connectivity has been a core principle of the Internet, supported for example by the Internet Architecture Board. Current Internet architectural documents observe that NAT is a violation of the End-to-End Principle, but that NAT does have a valid role in careful design.^[7] There is considerably more concern with the use of IPv6 NAT, and many IPv6 architects believe IPv6 was intended to remove the need for NAT.^[8]

Because of the short-lived nature of the stateful translation tables in NAT routers, devices on the internal network lose IP connectivity typically within a very short period of time unless they implement NAT

keep-alive mechanisms by frequently accessing outside hosts. This dramatically shortens the power reserves on battery-operated hand-held devices and has thwarted more widespread deployment of such IP-native Internet-enabled devices.^[*citation needed*]

Some Internet service providers (ISPs), especially in India, Russia, parts of Asia and other "developing" regions provide their customers only with "local" IP addresses, ~~due to a limited number of external IP~~ addresses allocated to those entities^[*citation needed*]. Thus, these customers must access services external to the ISP's network through NAT. As a result, the customers cannot achieve true end-to-end connectivity, in violation of the core principles of the Internet as laid out by the Internet Architecture Board^[*citation needed*].

- Scalability - An implementation that only tracks ports can be quickly depleted by internal applications that use multiple simultaneous connections (such as an HTTP request for a web page with many embedded objects). This problem can be mitigated by tracking the destination IP address in addition to the port (thus sharing a single local port with many remote hosts), at the expense of implementation complexity and CPU/memory resources of the translation device.
- Firewall complexity - Because the internal addresses are all disguised behind one publicly-accessible address, it is impossible for external hosts to initiate a connection to a particular internal host without special configuration on the firewall to forward connections to a particular port. Applications such as VOIP, videoconferencing, and other peer-to-peer applications must use NAT traversal techniques to function.

Specifications

IEEE ^[9] Reverse Address and Port Translation (RAPT, or RAT) allows a host whose real IP address is changing from time to time to remain reachable as a server via a fixed home IP address. In principle, this should allow setting up servers on DHCP-run networks. While not a perfect mobility solution, RAPT together with upcoming protocols like DHCP-DDNS, it may end up becoming another useful tool in the network admin's arsenal.

IETF ^[10] *RAPT* (IP Reachability Using Twice Network Address and Port Translation) The RAT device maps an IP datagram to its associated CN and OMN by using three additional fields: the IP protocol type number and the transport layer source and destination connection identifiers (e.g. TCP port number or ICMP echo request/reply ID field).

Cisco *RAPT* implementation is PAT (Port Address Translation) or overloading, and maps multiple private IP addresses to a single public IP address. Multiple addresses can be mapped to a single address because each private address is tracked by a port number. PAT uses unique source port numbers on the inside global IP address to distinguish between translations. The port number is encoded in 16 bits. The total number of internal addresses that can be translated to one external address could theoretically be as high as 65,536 per IP address. Realistically, the number of ports that can be assigned a single IP address is around 4000. PAT will attempt to preserve the original source port. If this source port is already used, PAT will assign the first available port number starting from the beginning of the appropriate port group 0-511, 512-1023, or 1024-65535. When there are no more ports available and there is more than one external IP address configured, PAT moves to the next IP address to try to allocate the original source port again. This process continues until it runs out of available ports and external IP addresses.

3COM U.S. Patent 6,055,236 (<http://www.google.com/patents?vid=6055236>) (Method and system for

locating network services with distributed network address translation) Methods and system for locating network services with distributed network address translation. Digital certificates are created that allow an external network device on an external network, such as the Internet, to request a service from an internal network device on an internal distributed network address translation network, such as a stub local area network. The digital certificates include information obtained with a Port Allocation Protocol used for distributed network address translation. The digital certificates are published on the internal network so they are accessible to external network devices. An external network device retrieves a digital certificate, extracts appropriate information, and sends a service request packet to an internal network device on an internal distributed network address translation network. The external network device is able to locate and request a service from an internal network device. An external network device can also request a security service, such as an Internet Protocol security ("IPsec") service from an internal network device. The external network device and the internal network device can establish a security service (e.g., Internet Key Exchange protocol service). The internal network device and external network device can then establish a Security Association using Security Parameter Indexes ("SPI") obtained using a distributed network address translation protocol. External network devices can request services, and security services on internal network devices on an internal distributed network address translation network that were previously unknown and unavailable to the external network devices.

Examples of NAT software

- Internet Connection Sharing (ICS): Windows NAT+DHCP since W98SE
- WinGate: like ICS plus lots of control
- iptables: the Linux packet filter and NAT (interface for NetFilter)
- IPFilter: Solaris, NetBSD, FreeBSD, xMach.
- PF (firewall): The OpenBSD Packet Filter.
- Netfilter Linux packet filter framework

See also

- AYIYA (IPv6 over IPv4 UDP thus working IPv6 tunneling over most NATs)
- Carrier Grade NAT
- Firewall
- Gateway
- Internet Gateway Device (IGD) Protocol: UPnP NAT-traversal method
- Middlebox
- Internet Protocol version 4
- NAT-PT
- Port forwarding
- Port triggering
- Private IP address
- Proxy server
- Routing
- Subnet
- port
- Teredo tunneling: NAT traversal using IPv6

References

1. ^ www.tcpipguide.com/free/t_IPNetworkAddressTranslationNATProtocol.htm (http://www.tcpipguide.com/free/t_IPNetworkAddressTranslationNATProtocol.htm)
2. ^ François Audet; and Cullen Jennings (January 2007) (text). *RFC 4787 Network Address Translation (NAT) Behavioral Requirements for Unicast UDP* (<http://www.ietf.org/rfc/rfc4787.txt>) . IETF. <http://www.ietf.org/rfc/rfc4787.txt>. Retrieved 2007-08-29.
3. ^ "Characterization and Measurement of TCP Traversal through NATs and Firewalls" (<http://nutss.gforge.cis.cornell.edu/pub/imc05-tcpnat/>) . December 2006. <http://nutss.gforge.cis.cornell.edu/pub/imc05-tcpnat/>.
4. ^ "Illuminating the shadows: Opportunistic network and web measurement" (<http://illuminati.coralcdn.org/stats/>) . December 2006. <http://illuminati.coralcdn.org/stats/>.
5. ^ "The Audio over IP Instant Expert Guide" (<http://www.tieline.com/Downloads/Audio-over-IP-Instant-Expert-Guide-v1.pdf>) . Tieline. January 2010. <http://www.tieline.com/Downloads/Audio-over-IP-Instant-Expert-Guide-v1.pdf>. Retrieved 2011-08-19.
6. ^ Larry L. Peterson; and Bruce S. Davie; *Computer Networks: A Systems Approach*, Morgan Kaufmann, 2003, pp. 328-330, ISBN 1-55860-832-X
7. ^ R. Bush; and D. Meyer; RFC 3439, *Some Internet Architectural Guidelines and Philosophy* (<http://www.ietf.org/rfc/rfc3439.txt>) , December 2002
8. ^ G. Van de Velde *et al.*; RFC 4864, *Local Network Protection for IPv6* (<http://tools.ietf.org/rfc/rfc4864.txt>) , May 2007
9. ^ <http://ieeexplore.ieee.org/iel4/6056/16183/00749275.pdf>
10. ^ <http://www3.ietf.org/proceedings/99nov/I-D/draft-ietf-nat-rnat-00.txt>

External links

- NAT-Traversal Test and results (<http://natatest.net.in.tum.de>)
- Characterization of different TCP NATs (<http://nutss.net/pub/imc05-tcpnat/>) – Paper discussing the different types of NAT
- Anatomy: A Look Inside Network Address Translators – Volume 7, Issue 3, September 2004 (http://www.cisco.com/en/US/about/ac123/ac147/archived_issues/ipj_7-3/anatomy.html)
- Jeff Tyson, HowStuffWorks: *How Network Address Translation Works* (<http://computer.howstuffworks.com/nat.htm/printable>)
- NAT traversal techniques in multimedia Networks (<http://www.newport-networks.com/whitepapers/nat-traversal1.html>) – White Paper from Newport Networks
- NAT traversal for IP Communications (<http://www.voiptraversal.com/EyeballAnyfirewallWhitePaper.pdf>) – White Paper from Eyeball Networks
- Peer-to-Peer Communication Across Network Address Translators (<http://www.brynosaurus.com/pub/net/p2pnat/>) (PDF) (<http://www.brynosaurus.com/pub/net/p2pnat.pdf>) – NAT traversal techniques for UDP and TCP
- <http://www.zdnetasia.com/insight/network/0,39044847,39050002,00.htm>
- RFCs
 - RFC 1631 (Status: Obsolete) - The IP Network Address Translator (NAT)
 - RFC 1918 - Address Allocation for Private Internets
 - RFC 3022 (Status: Informational) – Traditional IP Network Address Translator (Traditional NAT)
 - RFC 4008 (Status: Standards Track) – Definitions of Managed Objects for Network Address Translators (NAT)
 - RFC 5128 (Status: Informational) - State of Peer-to-Peer (P2P) Communications across Network Address Translators (NATs)
 - RFC 4966 (Status: Informational) - Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status

- *Speak Freely* End of Life Announcement (<http://www.fourmilab.ch/speakfree/unix/>) – John Walker's discussion of why he stopped developing a famous program for free Internet communication, part of which is directly related to NAT
- natd (http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/network-natd.html)
- SNAT, DNAT and OCS2007R2 (<http://www.cainetworks.com/support/training/snat-dnat-ocs.html>) – discussing the SNAT in Microsoft OCS 2007R2
- Alternative Taxonomy (Part of the documentation for the IBM iSeries)
 - Static NAT (<http://publib.boulder.ibm.com/infocenter/iseriess/v5r3/index.jsp?topic=/rzajw/rzajwstatic.htm>)
 - Dynamic NAT (<http://publib.boulder.ibm.com/infocenter/iseriess/v5r3/index.jsp?topic=/rzajw/rzajwdynamic.htm>)
 - Masquerade NAT (<http://publib.boulder.ibm.com/infocenter/iseriess/v5r3/index.jsp?topic=/rzajw/rzajwaddmasq.htm>)
- Network Address Translation - NAT (<http://blog.ipexpert.com/2009/09/07/network-address-translation-nat/>)
- Cisco Systems
 - Document ID 6450: How NAT Works (http://www.cisco.com/en/US/tech/tk648/tk361/technologies_tech_note09186a0080094831.shtml)
 - Document ID 26704: Network Address Translation (NAT) FAQ (http://www.cisco.com/en/US/tech/tk648/tk361/technologies_q_and_a_item09186a00800e523b.shtml)
 - White Paper: Cisco IOS Network Address Translation Overview (http://www.cisco.com/en/US/technologies/tk648/tk361/tk438/technologies_white_paper09186a0080091cb9.html)
 - Cisco IOS NAT Commands Cisco IOS commands (<http://www.cisco.com/univercd/cc/td/doc/product/software/ios113ed/cs/csprtd/csprtd11/csnat.htm>)
 - Animation Cisco NAT sample (<http://www.cisco.com/image/gif/paws/6450/nat.swf>)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Network_address_translation&oldid=481168651"

Categories: Network address translation | Internet architecture

-
- This page was last modified on 10 March 2012 at 14:58.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.



Speak Freely

End of Life Announcement

by John Walker
January 15th, 2004

An early
voice chat
program

The time has come to lower the curtain on *Speak Freely*. As of August 1st, 2003, version 7.6a of *Speak Freely* (Unix and Windows) was declared the final release of the program, and a banner was added to the general *Speak Freely* page and those specific to the Unix and Windows versions on the www.fourmilab.ch site announcing the end of life. No further development or maintenance will be done, and no subsequent releases will be forthcoming.

On January 15th, 2004 all *Speak Freely* documentation and program downloads, along with links to them on the site navigation pages, were removed from the www.fourmilab.ch site, and accesses to these files redirected to this document. On that date the **speak-freely** and **speak-freely-digest** mailing lists were closed and their archives copied to off-line storage and deleted from the site. In addition, the *Speak Freely Forum* will cease operation, along with the Echo and Look Who's Listening servers previously running at www.fourmilab.ch. Ports 2074 through 2076 will be firewall blocked for the [fourmilab.ch](http://www.fourmilab.ch) domain, with incoming packets silently discarded. As of January 15th, 2004, all queries, in whatever form, regarding *Speak Freely* will be ignored. An historical retrospective on the program may eventually be published on the site.

Questions and Answers

Why did you do this?

The time has come. *Speak Freely* is the direct descendant of a program I originally developed and posted to Usenet in 1991. The bulk of *Speak Freely* development was done in 1995 and 1996, with the Windows version designed around the constraints of 16-bit Windows 3.1. Like many programs of comparable age which have migrated from platform to platform and grown to encompass capabilities far beyond anything envisioned in their original design, *Speak Freely* shows its age. The code is messy, difficult to understand, and very easy to break when making even small modifications. The Windows and Unix versions, although interoperable, have diverged in design purely due to their differing histories, almost doubling the work involved in making any change which affects them both.

To continue development and maintenance of *Speak Freely*, the program requires a top to bottom rewrite, basing the Unix and

haha Windows version on an identical "engine," and providing an application programming interface (API) which permits other programs to be built upon it. I estimate the work involved in this task, simply to reach the point where a program built with the new architecture is 100% compatible with the existing *Speak Freely*, would require between 6 and 12 man-months. There is no prospect whatsoever that I will have time of that magnitude to devote to *Speak Freely* in the foreseeable future, and no indication that any other developer qualified to do the job and sufficiently self-motivated and -disciplined to get it done exists. In fact, the history of *Speak Freely* constitutes what amounts to a non-existence proof of candidate developers.

Even if I had the time to invest in *Speak Freely*, or another developer or group of developers volunteered to undertake the task, the prospects for such a program would not justify the investment of time.

What do you mean--isn't the Internet still in its infancy?

If you say so. The Internet, regardless of its state of development, is in the process of metamorphosing into something very different from the Internet we've known over the lifetime of *Speak Freely*. The Internet of the near future will be something never contemplated when *Speak Freely* was designed, inherently hostile to such peer-to-peer applications.

I am not using the phrase "peer to peer" as a euphemism for "file sharing" or other related activities, but in its original architectural sense, where all hosts on the Internet were fundamentally equal. ← Certainly, Internet connections differed in bandwidth, latency, and reliability, but apart from those physical properties any machine connected to the Internet could act as a client, server, or (in the case of datagram traffic such as *Speak Freely* audio) neither--simply a *peer* of those with which it communicated. Any Internet host could provide any service to any other and access services provided by them. New kinds of services could be invented as required, subject only to compatibility with the higher level transport protocols (such as TCP and UDP). Unfortunately, this era is coming to an end.

One need only read discussions on the *Speak Freely* mailing list and Forum over the last year to see how many users, after switching from slow, unreliable dial-up Internet connections to broadband, persistent access via DSL or cable television modems discover, to their dismay, that they can no longer receive calls from other *Speak Freely* users. The vast majority of such connections use Network Address Translation (NAT) in the router connected to the broadband link, which allows multiple machines on a local network to share the broadband Internet access. But NAT does a lot more than that.

A user behind a NAT box is no longer a peer to other sites on the Internet. Since the user no longer has an externally visible Internet

Protocol (IP) address (fixed or variable), there is no way (in the general case--there may be "workarounds" for specific NAT boxes, but they're basically exploiting bugs which will probably eventually be fixed) for sites to open connections or address packets to his machine. The user is demoted to acting exclusively as a client. While the user can contact and freely exchange packets with sites *not* behind NAT boxes, he *cannot* be reached by connections which originate at other sites. In economic terms, the NATted user has become a *consumer* of services provided by a higher-ranking class of sites, *producers* or *publishers*, not subject to NAT.

Need friends
mainly central
server

There are powerful forces, including government, large media organisations, and music publishers who think this situation is just fine. In essence, every time a user--they *love* the word "consumer"--goes behind a NAT box, a site which was formerly a peer to their own sites goes dark, no longer accessible to others on the Internet, while their privileged sites remain. The lights are going out all over the Internet. My paper, *The Digital Imprimatur*, discusses the technical background, economic motivations, and social consequences of this in much more (some will say tedious) detail. Suffice it to say that, as the current migration of individual Internet users to broadband connections with NAT proceeds, the population of users who can use a peer to peer telephony product like *Speak Freely* will shrink apace. It is irresponsible to encourage people to buy into a technology which will soon cease to work.

But isn't the problem really the lack of static port mapping, not NAT?

(If you don't understand this question, please skip to the next.)

Correct, but experience has shown that a large number of installed NAT boxes either cannot map an externally accessible port to an internal IP address and port, or those who install the boxes do not provide their customers adequate information to permit them to do this. Given the trend, discussed in the last question, toward confining individual Internet users to a consumer role, I believe fewer and fewer users will have the ability to statically map ports as time goes on.

has users were
hackers, right?

Isn't there some clever way to work around these limitations?

Not as far as I can figure out. As long as a NAT box only maps an inbound port to a local IP address when an outbound connection is established, I know of no way an Internet user can initiate a connection to a user behind a NAT box. With sufficient cleverness, such as the "NAT fix" in the 7.6a Unix version of *Speak Freely*, a user behind a NAT box can connect to one who isn't, but if both users are NATted (and that's the way things are going), the only way they could communicate would be through a non-NATted server site to which both connected, which would then forward packets between them.

UPnP
Security issue
Steve Gibson is
pro NAT

So why don't you just set up such a server?

Because no non-commercial site like mine could possibly afford the unlimited demands on bandwidth that would require. It's one thing to

could just
set up
connection?
Or do you need
open relays?

→ provide a central meeting point like a Look Who's Listening server, which handles a packet every five minutes or so from connected sites, but a server that's required to forward audio in real-time between potentially any number of simultaneously connected users is a bandwidth killer. The **www.fourmilab.ch** site has 2 Mbit/sec bidirectional bandwidth, about 50-75% of which (outbound) is typically in use serving Web pages. If we assume 1 Mbit/sec free bandwidth, then fewer than 70 simultaneous *Speak Freely* half-duplex GSM conversations would saturate this bandwidth, half that number if they're full-duplex. Besides, as soon as you set up such a server, within hours it would come under denial of service attacks mounted by malicious children and their moral and intellectual adult equivalents which would render the server unusable to legitimate users. Further, the existence of such server(s) would represent a single-point vulnerability which is the very antithesis of the design of the Internet and *Speak Freely*. Anybody who thinks through the economics and logistics of operating such a server on a *pro bono* basis will, I am confident, reject it on the same grounds I have. If you disagree, go prove me wrong!

But won't NAT go away once we migrate to IPv6?

(If you don't know what IPv6 is, please skip ahead to the next question.) First of all, any bets on when IPv6 will actually be implemented end-to-end for a substantial percentage of individual Internet users? And even if it were, don't bet on NAT going away. Certainly it will change, but once the powers that be have demoted Internet users from peers to consumers, I don't think they're likely to turn around and re-empower them just because the address space is now big enough. Besides, the fraction of users who care about such issues, while high among those interested in programs such as *Speak Freely*, is minuscule among the general public.

not now

Why January 15th, 2004?

January 1st would have made more sense, but I was out of town then, and I don't like to make major changes to the site while I'm on the road. There's no special significance to the date.

Can I go on using *Speak Freely*?

Certainly, *Speak Freely* is in the public domain; you can do anything you like with it. But as of that date I'm not having anything more to do with it.

Can I distribute copies of *Speak Freely* to other people?

Certainly; see the previous answer. You're free to distribute *Speak Freely* in source or binary form to anybody you like, post it on your Web site, etc. subject only to whatever governmental restrictions may apply to distribution of the encryption technology *Speak Freely* employs.

How will I be able to find people once your Look Who's Listening

No. While I cannot in good conscience encourage people to become new users of *Speak Freely* nor developers to invest time in working on it, the entire state of the program as of the final release will remain available indefinitely on [SourceForge](#) as separate CVS archives for the [Unix](#) and [Windows](#) versions. I will make no further additions to these archives, but others are free to download them for their own private development purposes and/or create new projects on [SourceForge](#) to develop derivative programs in whatever form they like.

Anything more to add?

It's been fun. Take care.

Speak Freely Afterlife Development Project

A [development project](#) has been registered on [SourceForge](#) with the goal of continuing development of *Speak Freely*. Anshuman Aggarwal and Johannes Pöhlmann, creators of this project, hope to attract developers to continue to adapt *Speak Freely* to the challenges it will face in the future. I am not involved in this project, but if you're interested in contributing to *Speak Freely*, please visit the project home page and volunteer for the effort.

The Digital Imprimatur

Fourmilab Home Page

server shuts down?

You can exchange IP addresses with people you wish to call via ICQ, instant messages, E-mail, chat systems, etc. If somebody wants to start a public Look Who's Listening server they're welcome, but history is not encouraging. While the Fourmilab LWL server has run continuously for 8 years, no other public server has lasted more than a year before disappearing.

How can I test without a public echo server?

Beats me. If the need is sufficient, perhaps somebody will set one up, but, as with LWL servers, they never seem to last very long.

Why all the dramatics of an "end of life" announcement?

The fate of most free software projects is "abandonware"--the developer loses interest, burns out, or becomes occupied with other projects and simply leaves the software as-is. In fact, this happened with *Speak Freely* a few years ago, and the consequences were distasteful. Unix workstation vendors routinely issue end of life announcements to inform customers that as of a given date, or software release, or new hardware platform, an existing product will no longer be supported. This gives those using that product time to weigh the alternatives and decide how best to proceed. Given that the Internet is in the midst of a structural change (widespread adoption of broadband with NAT) which destroys the 30 year old Internet architecture on which *Speak Freely* (and other true peer to peer programs) relies, I thought it more responsible to withdraw the program in this manner (while, as with a workstation end of life announcement, permitting satisfied users to continue to use it indefinitely) rather than let it wilt and die as the dark pall of NAT falls upon the Internet.

Don't you have an obligation to *whatever*?

Nope. Writing software and giving it away doesn't incur any obligation of any kind to any person. I've been working on this program off and on for more than 12 years. At my age (don't ask, but if I live as long as Bob Hope did, I'm more than half way to the checkered flag), the prospect of spending another five or ten years dreaming up clever countermeasures to an Internet that's evolving to make programs like *Speak Freely* impossible, in a climate where creating a tool some people find useful and giving it away only invites incessant malicious attacks upon it motivated solely by nihilism, for a shrinking user community forced to master the ever-growing complexity all of this requires does not appeal to me. Programs, like people, are born, grow rapidly, mature, and then eventually age and die. So it goes. If somebody disagrees and wants to step in, they're more than welcome, but such a person has yet to appear over the entire history of *Speak Freely*.

By taking down the *Speak Freely* site, aren't you throwing away all the work invested in the program?

Skype protocol

From Wikipedia, the free encyclopedia

Reading for fun

The **Skype protocol** is a proprietary Internet telephony network based on peer-to-peer architecture, used by Skype. The protocol's specifications have not been made publicly available by Skype and official applications using the protocol are closed-source.

The Skype network is not interoperable with most other VoIP networks without proper licensing from Skype. Digium, the main sponsor of Asterisk PBX released a driver licensed by Skype dubbed 'Skype for Asterisk' to interface as a client to the Skype network, however this still remains closed source.^[1] Numerous attempts to study and/or reverse engineer the protocol have been undertaken to reveal the protocol, investigate security or to allow unofficial clients.

Contents

- 1 Peer-to-peer architecture
- 2 Protocol
 - 2.1 Protocol detection
 - 2.1.1 Preliminaries
 - 2.1.2 Skype client
 - 2.2 Login
- 3 UDP
- 4 Obfuscation Layer
- 5 TCP
- 6 Low-level datagrams
 - 6.1 Object Lists
 - 6.2 Packet compression
- 7 Legal issues
- 8 Notes
- 9 References
- 10 External links

went down in 12/2010

↓

So they do use computers
to relay actual encrypted
calls

Peer-to-peer architecture

Skype was the first peer-to-peer IP telephony network,^[2] requiring minimal centralized infrastructure.^[citation needed] The Skype user directory is decentralized and distributed among the clients, or nodes, in the network.

The network contains three types of entities: supernodes, ordinary nodes, and the login server. Each client maintains a host cache with the IP address and port numbers of reachable supernodes.

Any client with good bandwidth, no restriction due to firewall or NAT, and adequate processing power can become a supernode. This puts an extra burden on those who connect to the Internet without NAT, as Skype may use their computers and Internet connections as third party for UDP hole punching (to directly connect two clients both behind NAT) or to completely relay other users' calls. Skype does not choose to supply server power with associated bandwidth required to provide the relay service for every client who needs it, instead it uses the resource of Skype clients.^[3]

ah yes the supernode

Supernodes relay communications on behalf of two other clients, both of which are behind firewalls or "one to many" Network address translation. The reason that relaying is required is that without relaying clients with firewall or NAT difficulties, the two clients would be unable to make or receive calls from other. Skype tries to get the two ends to

negotiate the connection details directly, but what can happen is that the sum of problems at both ends can mean that two cannot establish direct conversation.

The problems with firewalls and NAT can be

- The external port numbers or IP address are not derivable, because NAT rewrites them,
- The firewall and NAT in use prevents the session being received
- UDP is not usable due to NAT issues , such as timeout
- firewalls block many ports
- TCP through many to one NAT is always "outward only" by default - Adding Port Forwarding settings to the NAT router can allow receiving TCP sessions

Supernodes are grouped into *slots* (9-10 supernodes), and slots are grouped into *blocks* (8 slots).

Protocol

Signaling is encrypted using RC4; however, the method only obfuscates the traffic as the key can be recovered from the packet. Voice data is encrypted with AES.^[4]

The Skype client's application programming interface (API) opens the network to software developers. The Skype API allows other programs to use the Skype network to get "white pages" information and manage calls.

The Skype code is closed source, and the protocol is not standardized.^[5] Parts of the client use Internet Direct (Indy), an open source socket communication library.^[citation needed]

Protocol detection

Many networking and security companies claim to detect and control Skype's protocol for enterprise and carrier applications. While the specific detection methods used by these companies are often proprietary, Pearson's chi-squared test and stochastic characterization with Naive Bayes classifiers are two approaches that were published in 2007.^[6]

Preliminaries

Abbreviations that are used:

- SN: Skype network
- SC: Skype client
- HC: host cache

Skype client

The main functions of a Skype client are:

- login
- user search
- start and end calls
- media transfer
- presence messages
- video conference

Login

A Skype client authenticates the user with the login server, advertises its presence to other peers, determines the type of NAT and firewall it is behind and discovers nodes that have public IP addresses.

To connect to the Skype network, the host cache must contain a valid entry. A TCP connection must be established (i.e. to a supernode) otherwise the login will fail.

```

1.  start
2.  send UDP packet(s) to HC
3.  if no response within 5 seconds then
4.      attempt TCP connection with HC
5.      if not connected then
6.          attempt TCP connection with HC on port 80 (HTTP)
7.          if not connected then
8.              attempt TCP connection with HC on port 443 (HTTPS)
9.              if not connected then
10.                  attempts++
11.                  if attempts==5 then
12.                      fail
13.                  else
14.                      wait 6 seconds
15.                      goto step 2
16.  Success

```

After a Skype client is connected it must authenticate the username and password with the Skype login server. There are many different Skype login servers using different ports. An obfuscated list of servers is hardcoded in the Skype executable.

Skype servers are:

- | | | | |
|--------------------------|--------------------------|-------------------------|-------------------------|
| ▪ dir1.sd.skype.net:9010 | ▪ dir5.sd.skype.net:9010 | ▪ http1.sd.skype.net:80 | ▪ http5.sd.skype.net:80 |
| ▪ dir2.sd.skype.net:9010 | ▪ dir6.sd.skype.net:9010 | ▪ http2.sd.skype.net:80 | ▪ http6.sd.skype.net:80 |
| ▪ dir3.sd.skype.net:9010 | ▪ dir7.sd.skype.net:9010 | ▪ http3.sd.skype.net:80 | ▪ http7.sd.skype.net:80 |
| ▪ dir4.sd.skype.net:9010 | ▪ dir8.sd.skype.net:9010 | ▪ http4.sd.skype.net:80 | ▪ http8.sd.skype.net:80 |

Skype-SW connects randomly to 1-8.

On each login session, Skype generates a session key from 192 random bits. The session key is encrypted with the hard-coded login server's 1536-bit RSA key to form an encrypted session key. Skype also generates a 1024-bit private/public RSA key pair. An MD5 hash of a concatenation of the user name, constant string ("\nSkype\r\n") and password is used as a shared secret with the login server. The plain session key is hashed into a 256-bit AES key that is used to encrypt the session's public RSA key and the shared secret. The encrypted session key and the AES encrypted value are sent to the login server.

On the login server side, the plain session key is obtained by decrypting the encrypted session key using the login server's private RSA key. The plain session key is then used to decrypt the session's public RSA key and the shared secret. If the shared secret match, the login server will sign the user's public RSA key with its private key. The signed data is dispatched to the super nodes.

Upon searching for a buddy, a super node will return the buddy's public key signed by Skype. The SC will authenticate the buddy and agree on a session key by using the mentioned RSA key.

UDP

UDP packets:

```

!IP
!UDP
!Skype SoF
!Skype Crypted Data01

```


The Start of Frame (SoF) consists of:

- 1. frame ID number (2 bytes)
- 2. payload type (1 byte)
 - obfuscated payload
 - Ack/Nack packet
 - payload forwarding packet
 - payload resending packet
 - other

Obfuscation Layer

The RC4 encryption algorithm is used to obfuscate the payload of datagrams.

- 1. The CRC32 of public source and destination IP, Skype's packet ID are taken
- 2. Skype obfuscation layer's initialization vector (IV).

The XOR of these two 32-bit values is transformed to a 80-byte RC4 key using an unknown key engine.

A notable misuse of RC4 in Skype can be found on TCP streams (UDP is unaffected). The first 14 bytes (10 of which are known) are xored with the RC4 stream. Then, the cipher is reinitialized to encrypt the rest of the TCP stream.^[7]

TCP

TCP packets:



The Skype Init TCP packet contains

- the seed (4 bytes)
- init_str string 00 01 00 00 01 00 00 00 01/03

Low-level datagrams

Almost all traffic is ciphered. Each command has its parameters appended in an object list. The object list can be compressed.



Object Lists

An object can be a number, string, an IP:port, or even another object list. Each object has an ID. This ID identifies which command parameter the object is.

Object:	Number
	IP:Port
	List of numbers
	String
	RSA key

Object List	
List Size (n)	
Object 1	
.	
.	
Object n	

Packet compression

Packets can be compressed. The algorithm is a variation of arithmetic compression that uses reals instead of bits.

Legal issues

Reverse engineering of the Skype protocol by inspecting/disassembling binaries is prohibited by the terms and conditions of Skype's license agreement. However there are legal precedents when the reverse-engineering is aimed at interoperability of file formats and protocols.^{[8][9][10]} In the United States, the Digital Millennium Copyright Act grants a safe harbor to reverse engineer software for the purposes of interoperability with other software.^{[11][12]} In addition, many countries specifically permit a program to be copied for the purposes of reverse engineering.^[13]

Notes

- [^] Skype for Asterisk – Production Released! (<http://blogs.digium.com/2009/08/31/skype-for-asterisk-production-released/>) , By pengler, August 31st, 2009, Digium - The Asterisk Company
- [^] Page 11 in Salman A. Baset; Henning Schulzrinne (2004). "An analysis of the Skype peer-to-peer Internet telephony protocol". arXiv:cs/0412017v1 (<http://arxiv.org/abs/cs/0412017v1>) [cs.NI (<http://arxiv.org/archive/cs>.NI)].
- [^] Skype "3.3 Utilization of Your Computer" (http://www.skype.com/intl/en/legal/eula/#you_expect) , *End User License Agreement*, August 2010
- [^] Introduction Skype analysis Enforcing anti-Skype policies (http://www.ossir.org/windows/supports/2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf) , Skype uncovered Security study of Skype, Desclaux Fabrice, 7/11/2005, EADS CCR/ST/C
- [^] http://support.skype.com/en_US/faq/FA153/Which-protocols-does-Skype-use
- [^] Dario Bonfiglio et al. "Revealing Skype Traffic: When Randomness Plays with You," ACM SIGCOMM Computer Communication Review, Volume 37:4 (SIGCOMM 2007), p. 37-48 (<https://www.dpacket.org/articles/revealing-skype-traffic-when-randomness-plays-you>)
- [^] Fabrice Desclaux, Kostya Kortchinsky (2006-06-17). "Vanilla Skype part 2" (<http://www.recon.cx/en/f/vskype-part2.pdf>) . *RECON2006*. <http://www.recon.cx/en/f/vskype-part2.pdf>.
- [^] Sega vs Accolade, 1992
- [^] Sony vs Connectix, 2000
- [^] Pamela Samuelson and Suzanne Scotchmer, "The Law and Economics of Reverse Engineering", 111 *Yale Law Journal* 1575-1663 (May 2002) [1] (<http://www.yalelawjournal.org/pdf/111-7/SamuelsonFINAL.pdf>)
- [^] 17 U.S.C. Sec. 1201(f).
- [^] WIPO Copyright and Performances and Phonograms Treaties Implementation Act
- [^] In the French "intellectual property" law set, there is an exception that allows any software user to reverse engineer it. See *code de la propriété intellectuelle* (<http://legifrance.gouv.fr/affichCodeArticle.do?cidTexte=LEGITEXT0000060694&idArticle=LEGIARTI000006278920&dateTexte=20080329&categorieLien=cid>) (**French**). This law is the national implementation of a piece of EU legislation: Council Directive 91/250/EEC (<http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31991L0250:EN:NOT>) ,

since then repealed by Directive 2009/24/EC of the European Parliament and of the Council of 23 April 2009 on the legal protection of computer programs (<http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32009L0024:EN:NOT>)

which also has a very similar provision allowing reverse engineering/decompilation for the purposes of development and testing of independent but inter-operating programs).

References

- Salman A. Baset; Henning Schulzrinne (2004). "An analysis of the Skype peer-to-peer Internet telephony protocol". arXiv:cs/0412017v1 (<http://arxiv.org/abs/cs/0412017v1>) [cs.NI (<http://arxiv.org/archive/cs>.NI)].
- P. Biondi and F. Desclaux (March 3, 2006). "Silver Needle in the Skype" (<http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf>) . <http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf>.
- F. Desclaux and K. Kortchinsky (June 6, 2006). "Vanilla Skype - part 1" (<http://www.recon.cx/en/f/vskype-part1.pdf>) . <http://www.recon.cx/en/f/vskype-part1.pdf>.
- F. Desclaux and K. Kortchinsky (June 17, 2006). "Vanilla Skype - part 2" (<http://www.recon.cx/en/f/vskype-part2.pdf>) . <http://www.recon.cx/en/f/vskype-part2.pdf>.
- L. De Cicco, S. Mascolo, V. Palmisano (May 2007). "An Experimental Investigation of the Congestion Control Used by Skype VoIP." (http://c3lab.poliba.it/images/d/d2/Skype_wwic07.pdf) . *WWIC 07*. Springer. http://c3lab.poliba.it/images/d/d2/Skype_wwic07.pdf.
- L. De Cicco, S. Mascolo, V. Palmisano (December 9–11, 2008). "A Mathematical Model of the Skype VoIP Congestion Control Algorithm." (http://c3lab.poliba.it/images/2/22/Skype_voip_model.pdf) . *Proc. of IEEE Conference on Decision and Control 2008*. http://c3lab.poliba.it/images/2/22/Skype_voip_model.pdf.
- Dario Bonfiglio, Marco Melia, Michela Meo, Dario Rossi, Paolo Tofanelli (August 27–31, 2007). "Revealing Skype Traffic: When Randomness Plays With You" (<https://www.dpacket.org/articles/revealing-skype-traffic-when-randomness-plays-you>) . ACM SIGCOMM Computer Communication Review. <https://www.dpacket.org/articles/revealing-skype-traffic-when-randomness-plays-you>.

External links

- Repository of articles on Skype analysis (<http://www1.cs.columbia.edu/~salman/skype/>)
- Reversed skype protocol, this torrent contain sources in c++ (<http://thepiratebay.org/torrent/6442887>)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Skype_protocol&oldid=479339395"

Categories: Skype | VoIP protocols | Instant messaging protocols

-
- This page was last modified on 28 February 2012 at 19:04.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

(Only teacher to get undressed in class --- haha)

He thinks avgs lower than past
Some qu were hard
1st quiz is the hardest

This hands on is useful

Where are the delays? See in Traceroute

Today: End to End / NAT

Name changes / conversion

Remember what NATs change

- Not the data

Sometimes in real life

- the person at co is hidden
- you get anyone when you call a company

Prof: NAT not just since running out of addresses

Its security as well

②

He thinks will be some simpler tech will take over IPv6

Phone manuf. want to maintain control

- Have a lot of IPs
- Don't want people to log into their phones

If TCP/UDP we know what packet looks like

- Or set up circuit switching
- Use ICMP to set up virtual channel

NAT translates back

- Can use ports (part of TCP)
- don't need to use ports
- translates the ports

NAT to NAT

1. Central server
2. Port forwarding

③

If no part #

- Something else in protocol
 - That router knows about
-

End - to - End

Not dogmatic

Big 30 years ago
Life is a lot less clear

Not understand where + when it applies

Where to put functions

↳ Apps will do all the work anyway

No requirements up front in paper

But requirements are in the end points

Who knows what the app needs

④

At the time everyone built each component to be highly reliable
+ whole network

Like AT&T's phone company

Then people building in the middle

Would it still be reliable?

The Internet is like that!

Amazing from an economic + technical -POV

So do end-to-end checksums

- Different than Simon paper?
 - watch man's watch fell apart
- Could do smaller pieces
 - on file
- The checksum length was based on prob
of files not making
 - Not "reliability"
 - But probability that stuff fails

5

Though lots of care events - so many will happen

Some prob. when put pieces online

Timeouts

- iChat does not work to Singapore
- How long do you wait?
- It depends on app
- if talking to someone else in Boston it should be \downarrow than 'int'

Bluetooth headset encrypted

- App sends normal packet
- Put in place for security
- But easy to break - since simple protocol
- Esp if listen to keys at the beginning
- Instead should connect w/ wires to prevent keys over air
- Or \$W on phone to eavesdrop on calls

(6)

Qc Compression

which ~~the~~ protocol to use

Must compress before encrypting

End to end → put stuff at the right level

ATSC vs CISC

- works similar

Networks

- don't want to be dumb pipe

- want to add value - commodity

6.033
LI1 Protocols + Network
Layer

Network protocols

Dumb network \rightarrow Best effort

- tries to deliver
- won't guarantee
- App sets up reliable ^{delivery} guarantee

Protocol: set of agreements to define structure of conversation

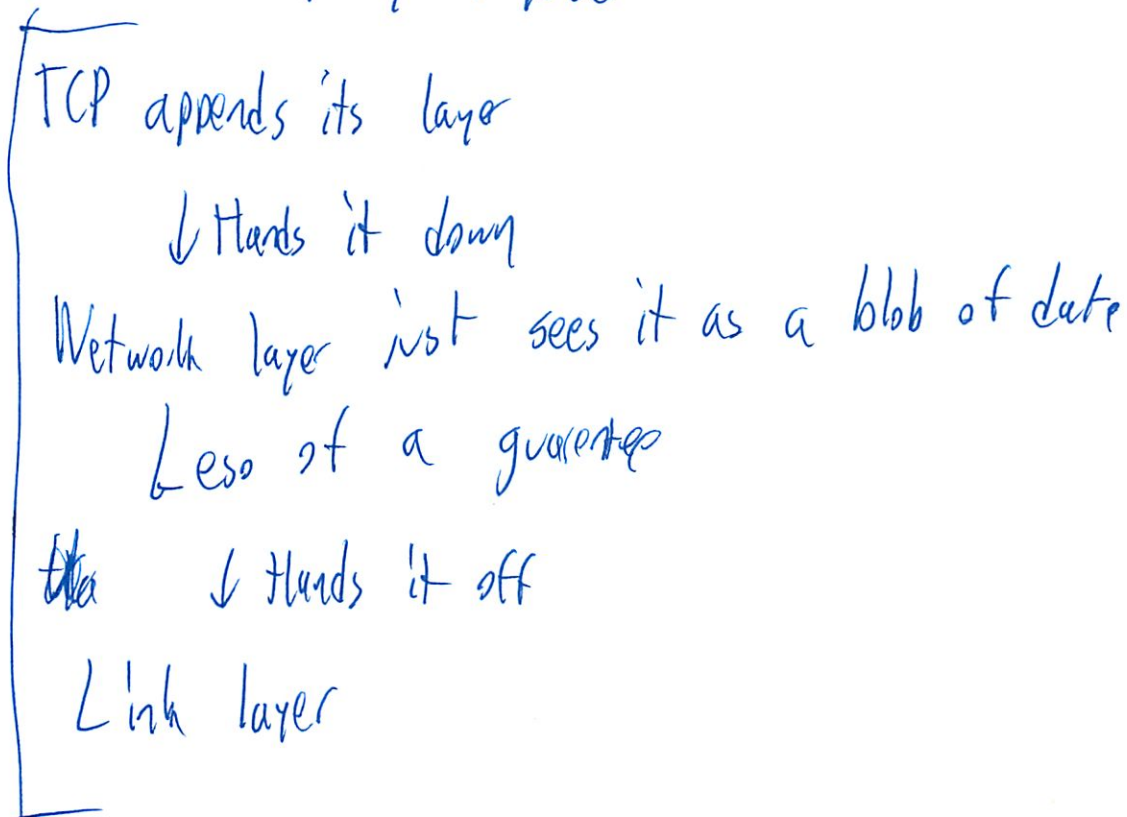
Can look at header of IP packets

- offset of which fragment it is
- TTL
- source / dest IP

Defined fully in an RFC

②

Protocols are typically layered



Then layers removed as item moves upwards

Layers are designed independently

NAT violates this rule slightly

How many layers do you need?

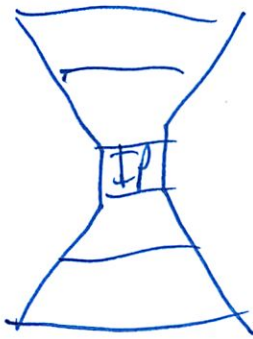
- big debate

- ISO : 7 layers is best

- internet : actually 4 layers

③

More like an hourglass



Wireshark

See actual packets

Take apart an email

(I've done this before...)

Next few lectures: going through the layers

Link layer - same as OSI

- framing
- error fixing
- Medium Access Control

④

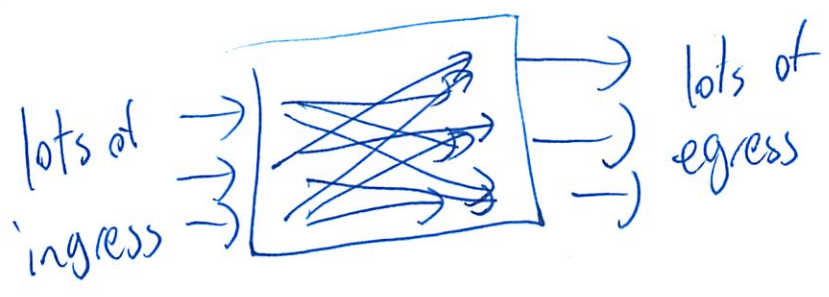
Network layer

traverse multiple hops across network

routing finding best path

forwarding send along to next link

forwarding table created by routing program



Won't talk about exact inside detail of routers

Basically

~~Header~~

- Look up DST in forwarding table
- Decrement TTL
- Recalc checksum
- Transmits packet down link

⑤

Hard to do this very fast
large rack switches 160 ~~GB/sec~~ 66/sec

Routing Problem

we generate a forwarding table

want shortest path
but no loops

At first was done by hand

Got way too complex to fast

Now hierarchical - each runs its own protocol

BGP

based on path vector protocol type

Each node ind. runs the protocol

it receives advertisements from its neighbors

Now it knows of its next door neighbors

Then everyone sends their routing table

Now it knows 2 hop connections

⑥

Then takes n rounds of advertising

But some issues

- 1) - loops
- 2) - if multiple paths
- 3) - if graph changes

1) It never accepts update when it is in the path

2) Some decision decision - least hops

3) Must run protocol continuously

- drop info if have not heard back in a while

If network is very big - then table might be too big

So have a series of domains / ~~autonomous~~ autonomous systems

- each is in charge of routing on the inside of the network

This cuts down size of routing tables / advertisements drastically

7

Demo of BGP counter

routeviews.org

- can look at its table
- and the size of the table
 - lots of bytes

- knows ~~some~~ about a bunch of tables entries

BGP visualization sites

- can see how networks changes over time (pretty cool)

Way more changes than I would have thought

No loops!

Hierarchical Routing

- Scaling
- Delegation

but Mobility is difficult

Paths can be suboptimal

⑧

But what is optimal?

< # hops

fastest

best biz relations

Can't move around the internet w/ same IP addr

Many open issues in routing

- Misconfiguration
- Flat addresses and scalable
- Routing in multipop WiFi networks
- Routing in P2P networks
 - Can play higher level, overlay routing protocol services

L11: Protocols and Network layer

Frans Kaashoek
6.033 Spring 2012

<http://web.mit.edu/6.033>

Some slides are from lectures by
Nick McKeown, Ion Stoica, Dina
Katabi, Hari Balakrishnan, Sam
Madden, and Robert Morris

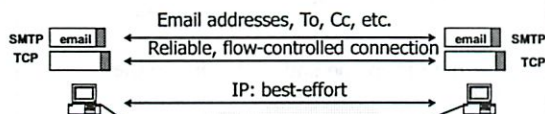


Internet: Best Effort

No Guarantees:

- Variable Delay (jitter)
- Variable rate
- Packet loss
- Duplicates
- Reordering
- Maximum length

End hosts implement everything else



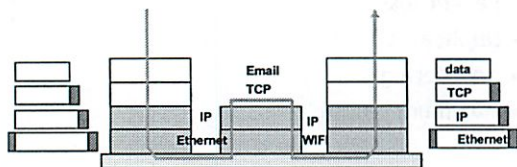
Protocol

- Defines the structure of a conversation
- Typical a sequence of messages, each with its own header
- Examples: DHCP, DNS, UDP, SMTP, TCP, IP, ...
- Internet protocols defined in text documents (RFCs)

vers	HLen	TOS	Total Length	
ID			Flags	FRAG Offset
TTL		Protocol	checksum	
SRC IP Address				
DST IP Address				
(OPTIONS)				(PAD)

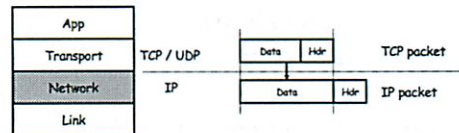
Layering of protocols

- Each layer adds/strips off its own header
- Each layer may split up higher-level data
- Each layer multiplexes multiple higher layers
- Each layer is (mostly) transparent to higher layers

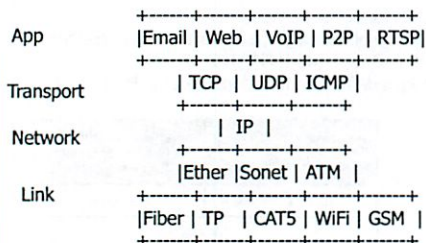


The Internet Stack

Protocol Stack



The Internet "Hour glass"



"Everything over IP, and IP over everything"

Link Layer



Problem:

Deliver data from one end of the link to the other

Need to address (6.02):

- Bits → Analog → Bits
- Framing
- Errors
- Medium Access Control

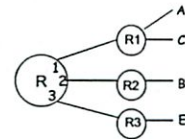
Network Layer:

finds a path to the destination and forwards packets along that path

- Difference between routing and forwarding
 - Routing is finding the path
 - Forwarding is the action of sending the packet to the next-hop toward its destination

Forwarding

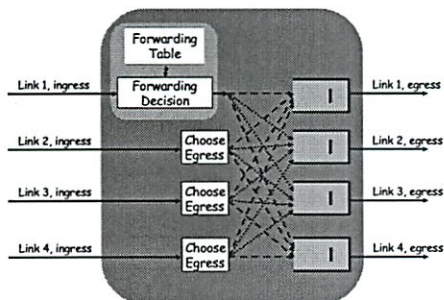
- Each router has a forwarding table
- Forwarding tables are created by a routing protocol



Forwarding table at R

Dst. Addr	Link
A	1
B	2
C	1
E	3

Inside a router



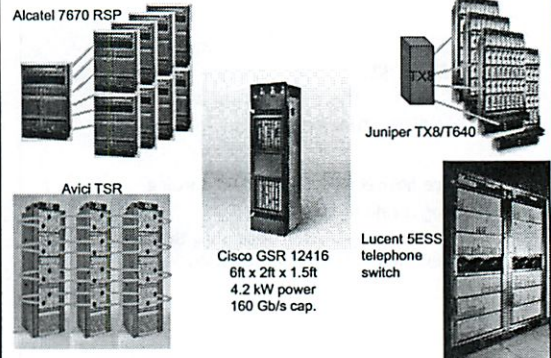
The IP Header

vers	HLen	TOS	Total Length	
ID		Flags	FRAG Offset	
TTL		Protocol	checksum	
SRC IP Address				
DST IP Address				
(OPTIONS)				(PAD)

Forwarding an IP Packet

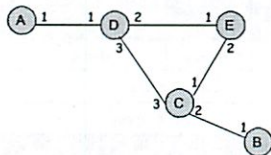
- Lookup packet's DST in forwarding table
 - If known, find the corresponding outgoing link
 - If unknown, drop packet
- Decrement TTL and drop packet if TTL is zero; update header Checksum
- Forward packet to outgoing port
- Transmit packet onto link

And switches today...



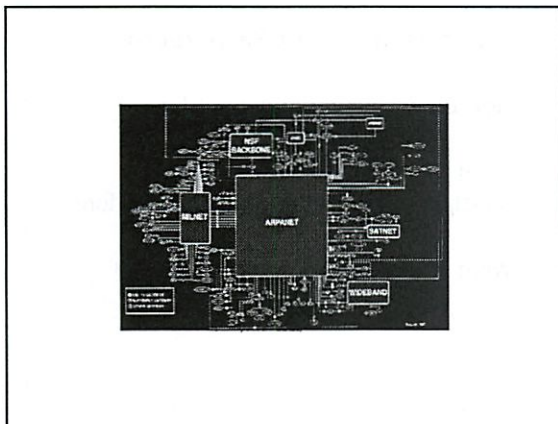
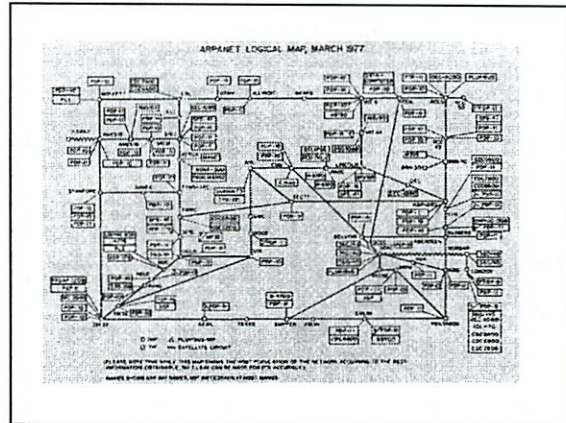
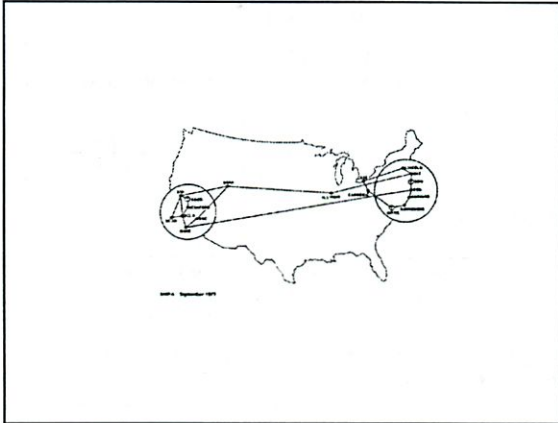
The Routing Problem:

- Generate forwarding tables



Goals: No loops, short paths, etc.





Path Vector Routing Protocol

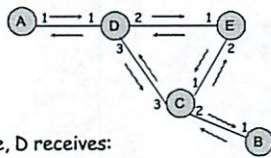
- Initialization
 - Each node knows the path to itself

For example, D initializes its paths

DST	Link	Path
D	End layer	null

Path Vector

- Step 1: Advertisement
 - Each node tells its neighbors its path to each node in the graph



For example, D receives:

From A:		From C:		From E:	
To	Path	To	Path	To	Path
A	null	C	null	E	null

Path Vector

- Step 2: Update Route Info
 - Each node use the advertisements to update its paths

D received: From A:		From C:		From E:	
To	Path	To	Path	To	Path
A	null	C	null	E	null

D updates its paths:

DST	Link	Path
D	End layer	null
A	1	<A>
C	3	<C>
E	2	<E>

Note: At the end of first round, each node has learned all one-hop paths

Path Vector

- Periodically repeat Steps 1 & 2

In round 2, D receives:

From A:		From C:		From E:	
To	Path	To	Path	To	Path
A	null	C	null	E	null
D	<D>	B		D	<D>

D updates its paths:

DST	Link	Path
D	End layer	null
A	1	<A>
C	3	<C>
E	2	<E>

Note: At the end of round 2, each node has learned all two-hop paths

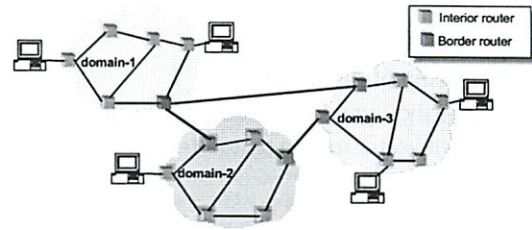
Questions About Path Vector

- How do we avoid permanent loops?
- What happens when a node hears multiple paths to the same destination?
- What happens if the graph changes?

Questions About Path Vector

- How do we ensure no loops?
 - When a node updates its paths, it never accepts a path that has itself
- What happens when a node hears multiple paths to the same destination?
 - It picks the better path (e.g., the shorter number of hops)
- What happens if the graph changes?
 - Algorithm deals well with new links
 - To deal with links that go down, each router should discard any path that a neighbor stops advertising

Hierarchical Routing

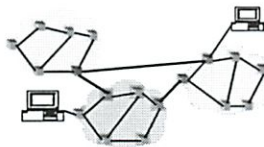


- Internet: collection of domains/networks
- Inside a domain: Route over a graph of routers
- Between domains: Route over a graph of domains
- Address consists of "Domain Id", "Node Id"

Hierarchical Routing

Advantage

- Scalable
 - Smaller tables
 - Smaller messages
- Delegation
 - Each domain can run its own routing protocol



Disadvantage

- Mobility is difficult
 - Address depends on geographic location
- Sub-optimal paths
 - E.g., in the figure, the shortest path between the two machines should traverse the yellow domain.

Routing: many open issues

- Misconfigurations between domains?
- Flat addresses and scalable?
- Routing in multihop WiFi networks?
- Routing in peer-to-peer networks?

Summary

- Protocols
- Layering of protocols
- Network layer: forwarding & Routing
 - Path-vector routing protocol

3/14

6.033 2011 Lecture 11: Layers and Network layer

Plan:

Protocol and Layers
 Link layer: 6.02
 Net layer: today
 E/E layer: Monday

How to solve problems from previous lecture?

Protocols
 Layers

Protocol

Examples: DHCP, DNS, UDP, SMTP, TCP, ...
 Two entities (peers) are talking.
 Protocol formally defines structure of conversation.
 Typically sequence of messages -- packets.
 Format:
 Specific set of message types.
 What does each bit of the message mean?
 [example: ethernet paper, dst, src, data, checksum]
 Rules for what happens next, state machines.
 Semantics: what does it mean?
 Often you can learn all you need from the formats...

Layers:

Intuition: protocols nest.
 Inner protocols building blocks for outer ones.
 Let's formalize that way of organizing multiple protos.
 Choose and define a useful protocol. [box <--> box]
 Define s/w interface so other layers can use it. [stack up]
 It may in turn use more primitive layers. [stack down]
 Can build up functionality this way.
 But use modules, abstraction to control complexity.
 Hard part: choosing useful layer boundaries.

6.033 layer model:

[draw stacks for host, switch, host]
 Physical: analog waveforms -> bits.
 Link: bits -> packets, single wire.
 Network: packet on wire -> packet to destination.
 End-to-end: packets -> connections or streams.
 Application.
 physical almost always tightly bound to Link.
 And application isn't a generally useful tool.
 Note: layer may have many clients
 multiple apps using e2e, multiple e2e using net
 need a way to multiplex them
 Note: net layer may use multiple links
 So the real picture in one host
 app1 app2 app3
 TCP UDP
 IP
 Eth WiFi
 Layers == outline of data networking topic.

Stack of layers:

Repeated scheme for layer interaction
 Each layer adds/strips off its own header.
 Encapsulates higher layer's data as "payload".
 [add to pkt diagram; interior header &c]
 Each layer may split up higher layer's data.
 [stream split into payloads of packets]
 Each layer multiplexes multiple higher layers.
 [put protocol # field into packet]
 Each layer is (mostly) transparent to higher layers.
 data delivered up on far side == data in

Demo: layers in action

```
wireshark
select interface wifi
capture all packets
send email
stop capture
filter traffic by src: ip.src == 192.168.1.108
look at trace
SMTP, TCP, etc.
```

Network layer

Forwarding -- sending data over links according to a routing table
 Routing -- process whereby routing tables are built

Forwarding -- mechanical. Just perform a lookup in a table.

Pseudocode:

```
forwarding_table t

net_send(payload, dest, e2eprot):
    pkt = new packet(payload, dest, e2eprot)
    net_handle(pkt)

net_handle(pkt):
    if (pkt.dest == LOCAL_ADDR):
        e2e_handle(pkt.payload, pkt.e2eprot)
    else:
        link_send(t[p.dest].link, pkt)    // table lookup
```

Routing -- compute the forwarding table

How to compute forwarding table? Manually -- not scalable.

Centrally -- not a good idea (why?)

- need a routing algorithm to collect
- collection requires many messages

- hard to adapt to changes

Path Vector Algorithm -- Distributed

Each node maintains a forwarding table T, with:

Dest	Link	Path
------	------	------

Two steps:

advertise (periodic)
send T to neighbors

integrate(N, neighbor, link) -- on receipt of advertisement from neighbor
merge neighbor table N heard from neighbor on link into T

Merging:

for each dest d w/ path r in N:
if d not in T, add (d, link, neighbor ++ r) to T
if d is in T, replace if (neighbor ++ r) is shorter than old path

Example:

(If everybody picks best path to every dest, you can see that for a network with most distant nodes separated by N hops, in N rounds everyone

Q: what is the purpose of keeping the path in the table?

Problems:

- permanent loops?
- won't arise if we add a rule that we don't pick paths with ourselves in them; this is what we need the path for!
- temporary loops -- arise because two nodes may be slightly out of date
example
- soln: add send count -- "TTL" -- to packet
- failures / changes -- repeat advertisements periodically,
remove paths in your table that aren't re-advertised
(e.g., a path P that begins with router R should be in the next advertisement from R.)
- graph changes -- same as failures

How does this work on the Internet:

At first, internet was a small network like this

Show evolution slides

What is the problem with using path vector here?

Network is huge

> 1 B nodes on network

Even if we assume most of those are computers that connect to only their local router (so don't really need to run the path vector

protocol), there are

Each router needs to know how to reach of these billions of computers

With pure path vector, each node has a multi-billion entry table (requiring gigabytes of storage)

Each router has to send these gigabyte tables to each of its neighbors; millions of advertisements propagating around. Disaster.

Solution: hierarchical routing

Subdivide net into areas; with multiple levels of routing

One node representative of each area; perform path vector at area level. Within each area, free to do whatever. (For example, use more hi

Demo: <http://www.routeviews.org/>

telnet route-views.routeviews.org

logged into a router, running view

show bgp 18.0.0.1

area.name

E.g., 18.7.22.69 -- this is mit.edu, area 18, which corresponds to AS 3 in BGP

<http://bgp.potaroo.net/cidr/autnums.html> list all AS

Internet routers running -- BGP -- advertise prefixes of these address

Show advertisements (e.g., "18.*.*.*") 17.1.*.*

show bgp 18.26.4.9

same table (only knows about 18).

show ip bgp sum

size of table

<http://bgplay.routeviews.org/>

query: 18..0.0/8

start 2/2/2011

end 3/8/2011

174: cognet

1239: sprint

3356: level

10578 (gigapop-NE, harvard)

6.033: Computer Systems Engineering

Spring
2012

Read 3/11

Home / News

Schedule

Submissions

General Information

Staff List

Recitations

TA Office Hours

Discussion / feedback

FAQ

Class Notes Errata

Excellent Writing Examples

2011 Home



Preparation for Recitation 12

Update: A new version of the Wide-Area Internet Routing notes is available here. This is an update to the notes in the course packet. It is similar in content to the previous notes, so it's okay if you've already read the old version. You may skip the appendices in these notes, and skip or skim sections 3.3 through 3.5.

Read *An Introduction to Wide-Area Internet Routing* (reading #11 in course packet)

Make sure you've read Section 7.4 of the textbook. For the paper, you'll specifically want a very good understanding of Section 7.4.2 (the path vector protocol).

This paper was written specifically for 6.033, so ought to go down easier than some of the research papers we've been giving you. Nonetheless, some of the standard reading tactics apply.

- Notice that the paper is full of acronyms and technical terms like "route reflectors" and "confederations." Which if any do you actually need to remember or understand in order to get the main idea of the paper? Ignore the others. For starters, you might keep an eye out for BGP, IP, AS, customer, provider, peering, transit.
- Start by reading the abstract and conclusion (section 3.6). The "take home points" may not make complete sense at first, but they will tell you what to look out for as you read the paper.
- Next go through the introduction. And take another look at 3.6: its points should make more sense now.
- Section two can be thought of as specifying the "requirements" for wide area routing: it discusses the kind of things network providers want to be able to do. This requirements discussion spills over to the beginning of Section 3 (up to but not including 3.1) and is relatively understandable.
- Sections 3.1 and on get into the details of how the requirements are met. Many of those details are less important; you might ignore them until you've had a chance to mull over the higher level issues for a while.

As you finish, consider the obvious general question:

- Why was this paper assigned? What useful information does it convey beyond that given in the text section 7.4.2?

And here are some specific ones:

- What are peering and transit relationships, and how are they different?
- Why do providers so dislike carrying other providers' packets? How does it hurt the provider to do so?
- What is a path-vector protocol? Why does the path-vector protocol in 7.4.2 look at hops over links, while BGP looks at hops through ASs?
- How much fault isolation do you get from BGP? What happens to the routing in the rest of the network, short and long term, if some router turns off? What is the worst that can happen if some router starts misbehaving (sending incorrect information)?
- Does routing have to be this complicated? Is it complex because of complex requirements, or because of bad design choices?
- Routers use the BGP protocol to set up routes by talking to each other. But they talk to each other using TCP, which relies on the presence of routes. Doesn't this create a chicken and egg problem?

Some students have found the [Cisco technology handbook](#) useful for understanding BGP.

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

[Top](#) // [6.033 home](#) //

(5 min later)

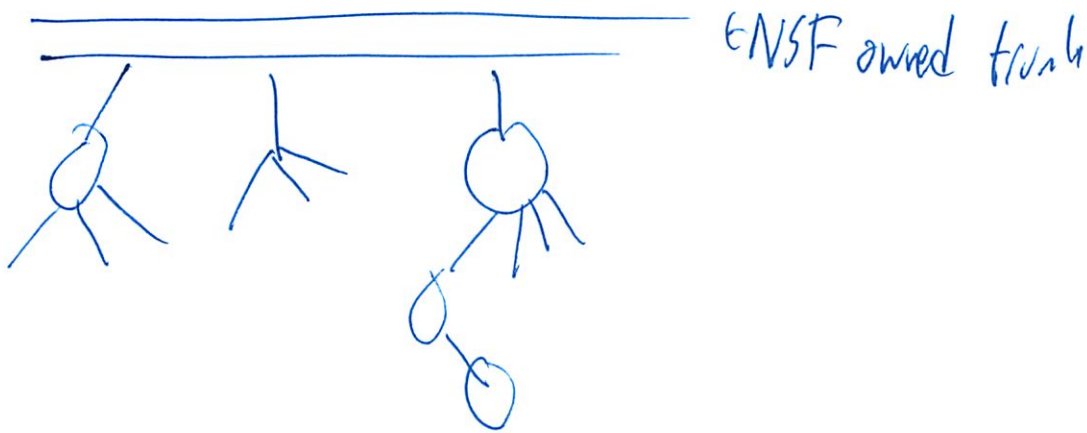
The Internet grew out of ARPAnet

- would not have got started w/o gov

NSFnet started free network

Hierarchy

Route 18. *. *. *. somewhere



Not how it works now

A bunch of independent companies competing

Tier 1

" 2

" 3

②

(Taled about copyright - no tech sense
- only business sense)

(Talking about net neutrality)

(He's not explaining it well)

(Going into peering)

(I'm ~~not~~ answering all the qrs ...)

If does not know anything it goes up

Internal
- like DHCP table

IP	Mac

External

on

IP	As
8.x.x.x	3

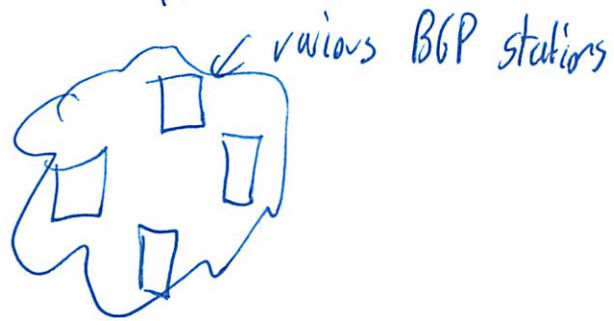
actually
8.0.0.0/8

(corrected him)

Dynamically decides where to send packets

③ BGP ~~all~~ operates on top of TCP

Internal BGP \rightarrow iBGP



What if you run BGP on your desktop

Pakistan YouTube Diagnostics messy

Tension: too much 'inspection', everyone will encrypt
Packet will be too big

ISPs try to keep you inside the network

6.033 Tutorial
Design Reports and Other Advice

Attendees

Travis Grusecki, TA for 6.033 Sections 1 and 2

Section 1 students and their writing instructor: Dave Custer

Section 2 students and their writing instructor, Linda Sutliff

Friday, March 16, 2012, 1:00 p.m. to 2:00 p.m. and 2:00 p.m. to 3:00 p.m., Room 36-153

Agenda

Topic	Lead
A. Design analysis and discussion	Students for first five minutes
B. Debriefing on DP-1	Custer, Grusecki and Sutliff
C. Complete draft design analysis form	Students
D. Context/primary objective	Custer

Lots of Oreo's

What's important of Oreo?

- people who only eat top + bottom

- Readers like introduction
info
bring it together

- Refer back to old info
Add new info

- Make sure to include

1. problem + context

2. Design \rightarrow Define it + Describe big picture
and structure
Need the big picture

- Don't ship since world limit

② Travis

Feedback online now

Technical grade is an advisory grade

If comments say include this
- include it

Don't view it as a grade
Graphics are good!

Put more in > 2-3

Larry will grade ~~with~~ final

Data structure important

- really specify what goes in there
- how big they grow

Think audience is a SW engineer manager

Show you have solved all technical problems

Cleaning up: garbage collection

no write ans, but should do something

③

Needs to support other things back does

Lie hardnames

there is no 1 abs path name

Fewer comments better

Everyone has fixable design

- few minor tweaks

Some of the excellent examples are too long

$\pm 100 - 200$ words ok

You are the expert now. The problem is condensing it - not like 4th grade where you didn't know what to talk about

Be more upfront in intro

Prefer 3rd party

Frame as a report

- you've been awarded contract - now design it

4

Some papers outstanding

But others lost pts for silly reasons

- like style spec
- cover page!

●

- draw graphics professionally
- Label "Figure 1" w/ a caption
- explain your points in the caption

You tell them what you will tell them

You tell them

✓ You tell them what you told them

~~brackets not in place~~

How to Frame

In order to (problem)

← wherever in doubt
in abstract

This paper describes a design ← at end
Optimized for (context)

(5)

Fill in Instant Paper part 3 - overall rationale

- why we built it?
- Or how we designed it?

Track provenance w/ minimal interruption
to current system

- I think pretty well described in the assignment

Engineering is about trade offs

Need a context in which to make these tradeoffs

The context was not emphasized at start of design

Context could even be done afterwards

Use case

Scenario

- come up w/ a use for a design

what tradeoffs are acceptable for a design

⑥
Other peoples

- hospital tracking info
- want fast + usability
 - not disc space
- Completeness
 - app has to do stuff

Describe your API implementation

3. Design Analysis and Discussion What is the overall rationale for your design?

Workload Analysis: Describe how your system performs on the workloads given in Section III.2 of the assignment.

List tradeoffs and their consequences

Design Tradeoff	Impact	Benefit

Explain the reasoning behind your system design. How do elements of your design provide good performance on your workloads? Do you see any limitations to your design that might crop up, perhaps as the system scales?

4. Conclusion: Does your solution solve the design problem? How? Is there any other work necessary in order to implement the design?

Part 2 – DP1 Instant Paper

Use this form as a way to organize the information you have been working with on DP1. You may want to complete these sections out of order. For example, many students find it easiest to complete sections 3 and 4 first, then return to section 1.

(Title Page) Describe the subject and scope of your design (aim for 15 words or fewer)

1. Introduction and Overview

What technical problem are you solving? What challenges do you face as a systems designer. (Answer in 2-3 sentences)

What is your overall strategy? (What were the major choices you made in this design?)

What are the chief characteristics of the resulting system? Complete the following sentences: "The goal of this design is to provide...We accomplish this goal by..."

2. Design Description How does your solution satisfy the constraints of the design problem?

6.033, Spring 2012

16 March, 1:00/2:00 (circle one)

Design Proposal 1(DP1) Report

Please leave this with your TA. Neither your name nor any other identifying information should be written on this sheet. Circle one of the options below:

I **DO** have a clear idea of how to write the DP1 Report

I **DO NOT** have a clear idea of how to write the DP1 Report

Is there any additional information you could have used in this session?

Tear Here-----Tear Here-----Tear Here-----Tear Here

KEEP THIS INFORMATION ON WHERE TO GET ADDITIONAL HELP

The links to IEEE standards:

1. Citations: <http://www.ieee.org/documents/ieeecitationref.pdf>
2. Style Manual: <http://www.ieee.org/documents/stylemanual.pdf>

If you have additional questions about this assignment:

1. E-mail the Writing Program lecturer running your tutorial section: All writing program lecturers will have office hours to address DP1 issues.
 - Section 1 (Except Varley and Wong): Dave Custer (custerspiral@gmail.com)
 - Section 2 (plus Varley and Wong): Linda Sutliff (lsutliff@mit.edu)
2. Feel free to e-mail your TA as well:
 - Travis Grusecki: (travisrg@MIT.EDU)
3. Contact the CI-M co-coordinators for the class: Jessie Stickgold-Sarah, Jessie@mit.edu or Don Unger, donunger@mit.edu

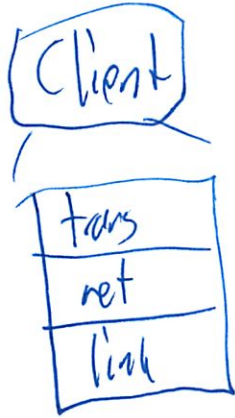
L12 Peer to Peer Systems

3/19

Peer to Peer systems → no central server

- Overlay model
 - treat transport layer as link layer for other network
- backup problem
- finding material

Client/Server



[No recitation Thur]

Examples

X-Window

DNS

Downsides

- Single pt of failure
- can have multiple servers - but expensive
- lots of management overhead to run
- no obvious central server

②

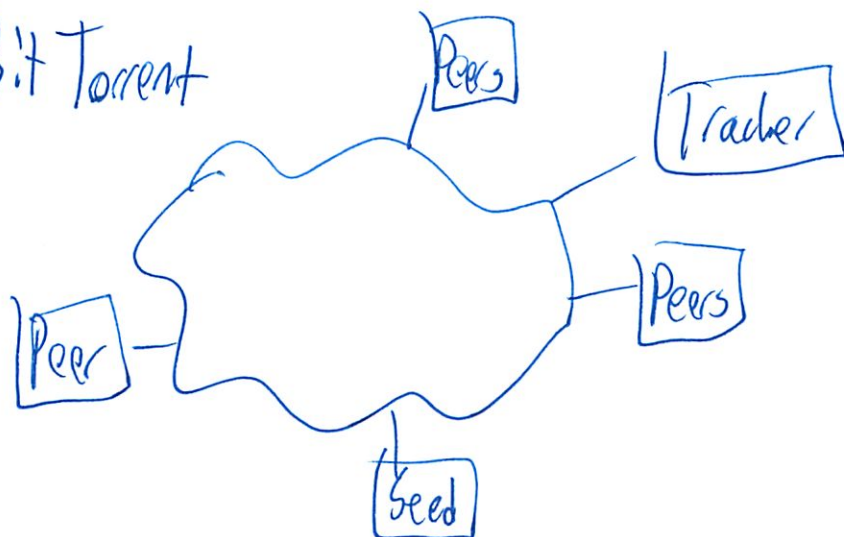
- lack ability to aggregate clients

Goals

- No central server
 - every machine is a server
- Every machine plays every machines' role
- How to find other nodes, data
- How do ya divide data across machines
- How to ~~make~~ do fault tolerances
 - maintain data online
- No central authority
- Some are still open problems

Example

Bit Torrent



③

Goal: distribute big files

Old fusion way - just download from seeder

New way

Seeder issues a pointer to tracker

Tracker maintains who has what pieces of files

Typically split into $\frac{1}{4}$ MB pieces

When peer finishes, becomes a seeder

Crypto hash of each piece

- to verify got right data

~~Which~~ Download

Which pieces to download?

- in order - but then ^{at start} everyone tries to get piece 1
- rarest - encourages quick ~~download~~ dist. of data
So more fault tolerant faster
- random

Can subdivide pieces

- Could download same blocks from multiple peers
↳ similar to how map reduce deals w/ strider

④

So BT does

1. Random list
2. Then rarest
3. For last block - parallel

He downloaded BitTorrent

↳ but was a leecher

Is this system really distributed?

The tracker is like a server!

(can have multiple trackers in 1 file)

But could a tracker be distributed?

↳ DMT

- tracker functionality distributed to peers
- put (URL, IP) into distributed hash table
- (can then ask get(URL))

⑤

Don't really want every peer to know about every other peer

- too much data.

So each machine is responsible for a particular part of the table

Chord protocol

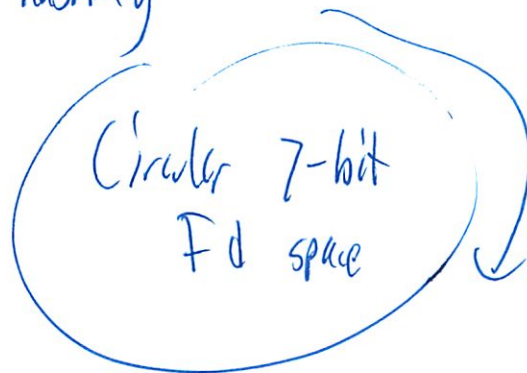
Efficient $O(\log N)$ messages per lookup

Scalable $O(\log N)$ state per node

Robust can survive many nodes going down

STHA-1 The key and the nodes (separately)

Consistent hashing



A key is stored at its successor

6

(never heard about this before)

Can do a basic lookup



ask who is the nodes successor

Will visit on avg $\frac{n}{2}$

Just give each node more state \rightarrow finger table
 $O(\log n)$

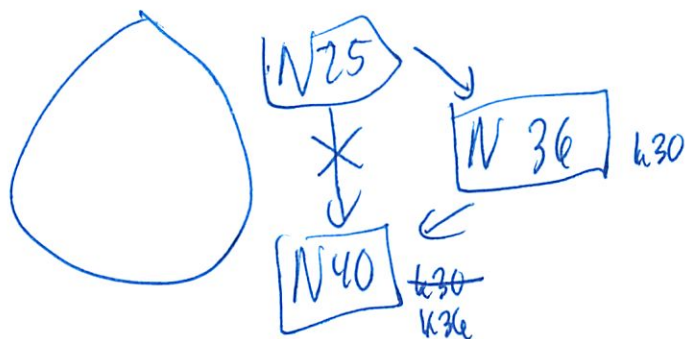
$\frac{1}{2}$
 $\frac{1}{4}$
 $\frac{1}{8}$
 $\frac{1}{16}$
 $\frac{1}{32}$
 $\frac{1}{64}$
 $\frac{1}{128}$

around the circle

Finger i points to successor of $n + 2^i$
Looks for highest node that proceeds
Every hop is half of previous hop
So $\log(N)$ hops

⑦

Joining linked list insert
how to add items?



Update finger ~~pointer~~ pointers in by
(an always find the nodes)
Just need to refresh the pointers

Incorrect lookup

N36 might not know N13 so goes to N138

So instead track next n successors

So don't overshoot

How many successors to maintain?

Assume $1/2$ nodes fail

$$P(\text{all successors dead}) = (1/2)^n$$

$$\text{So } P(\text{no broken}) = (1 - (1/2)^n)^n$$

⑧

(all always walk around ring w/ r successors
if all fingers are done)

This is w/o any servers

Some more various design issues

- concurrent joins
- locality
- heterogeneous node
- dishonest node

(Research chord more)

Peer-to-peer systems

6.033 Lecture 12, 2012

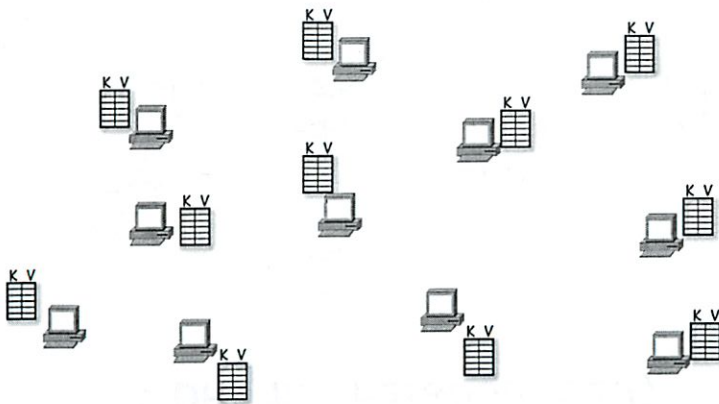
Frans Kaashoek

DP1: deadline Thursday 5p

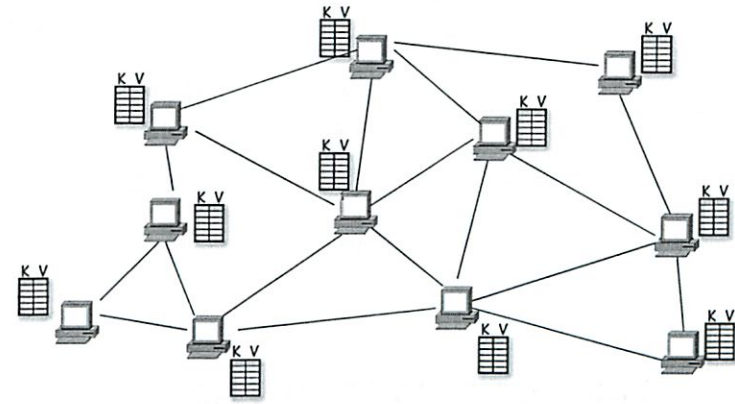
A torrent file

```
{
  'announce': 'http://bttracker.debian.org:6969/announce',
  'info':
  {
    'name': 'debian-503-amd64-CD-1.iso',
    'piece length': 262144,
    'length': 678301696,
    'pieces':
    '841ae846bc5b6d7bd6e9aa3dd9e551559c82abc1...d14f1631d
    776008f83772ee170c42411618190a4'
  }
}
```

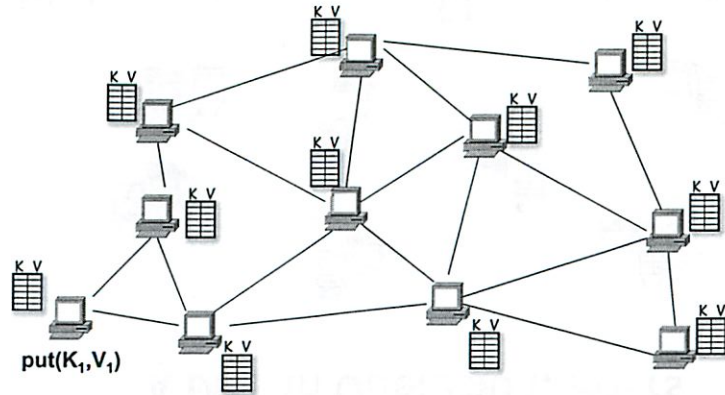
A DHT in Operation: Peers



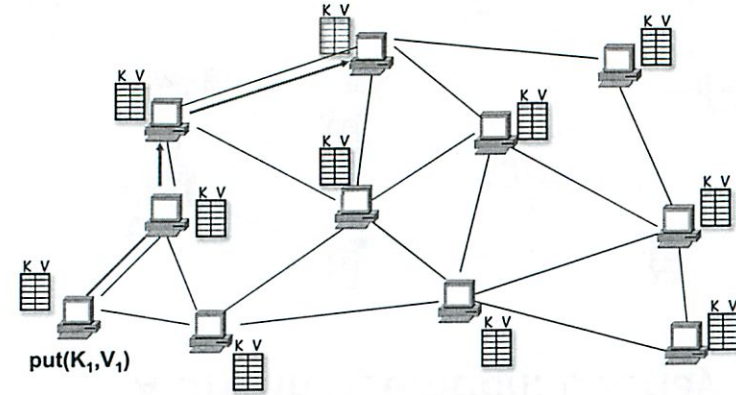
A DHT in Operation: Overlay



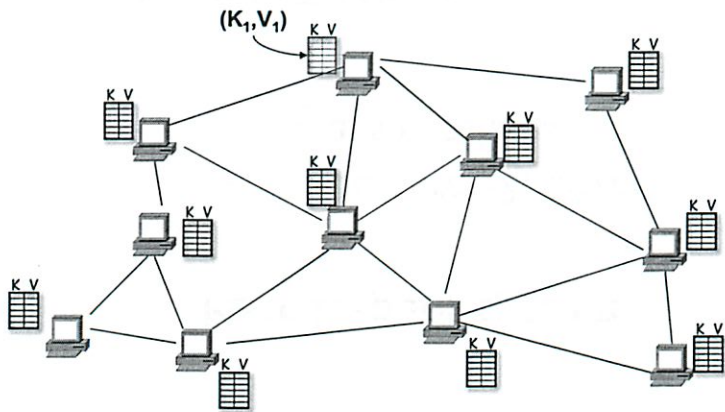
A DHT in Operation: put()



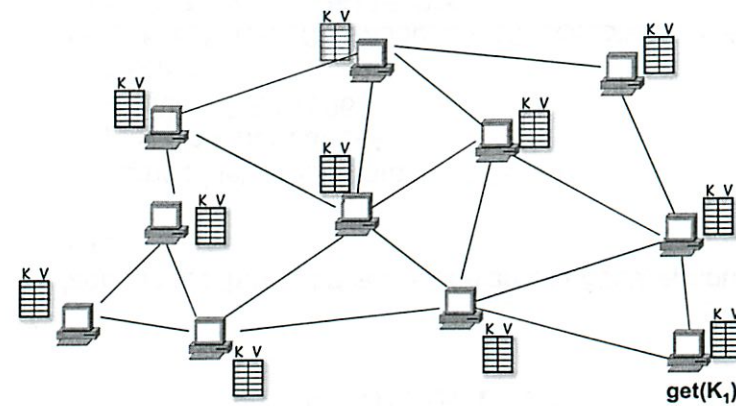
A DHT in Operation: put()



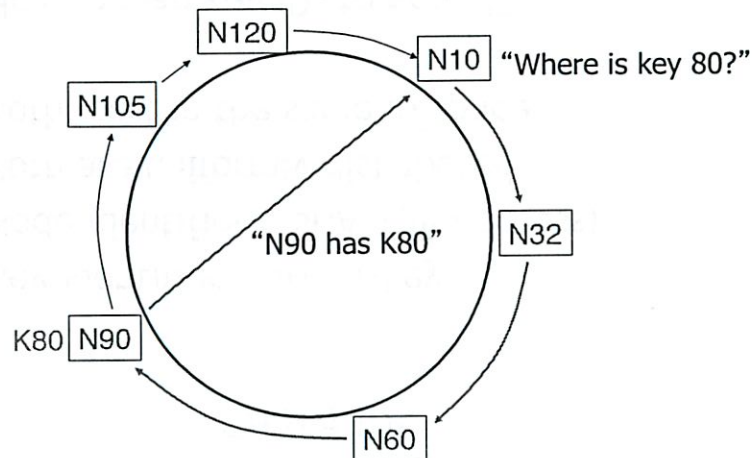
A DHT in Operation: put()



A DHT in Operation: get()



Basic lookup



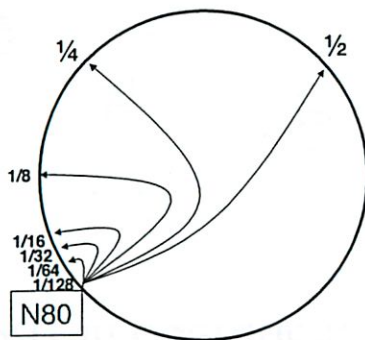
Simple lookup algorithm

```

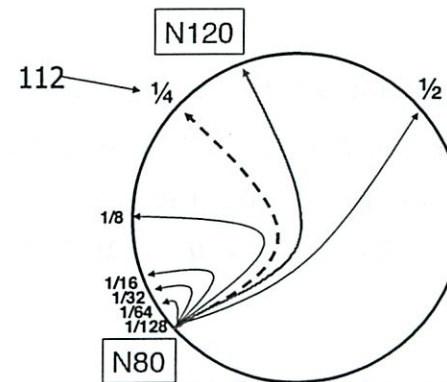
Lookup(my-id, key-id)
  n = my successor
  if my-id < n < key-id
    call Lookup(id) on node n // next hop
  else
    return my successor      // done
    
```

- Correctness depends only on successors

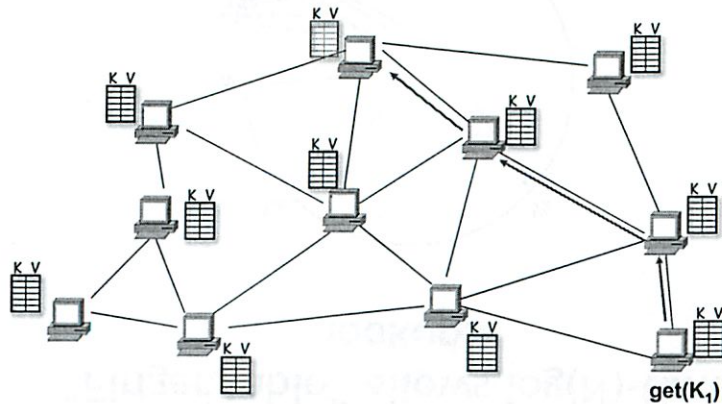
"Finger table" allows $\log(N)$ -time lookups



Finger i points to successor of $n+2^i$



A DHT in Operation: get()



Challenge: nodes join and leave

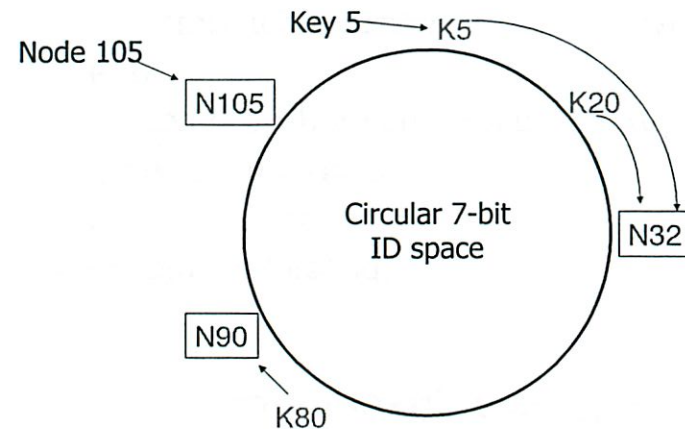
Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space
- How to map key IDs to node IDs?

Chord properties

- Efficient: $O(\log(N))$ messages per lookup
 - N is the total number of servers
- Scalable: $O(\log(N))$ state per node
- Robust: survives massive failures

Consistent hashing



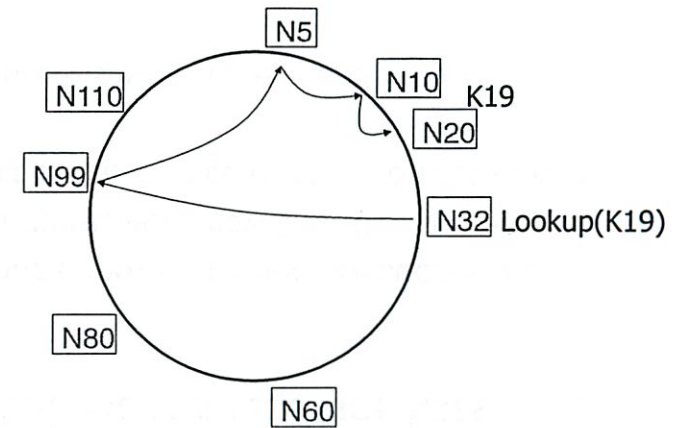
A key is stored at its successor: node with next higher ID

Lookup with fingers

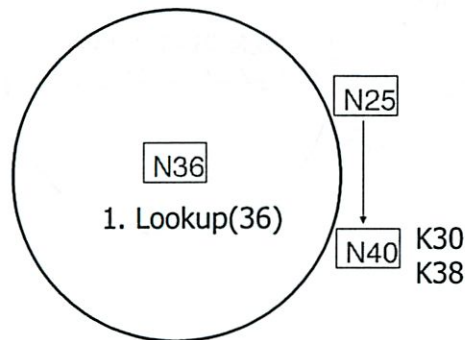
```

Lookup(my-id, key-id)
  look in local finger table for
    highest node n s.t. my-id < n < key-id
  if n exists
    call Lookup(id) on node n    // next hop
  else
    return my successor        // done
    
```

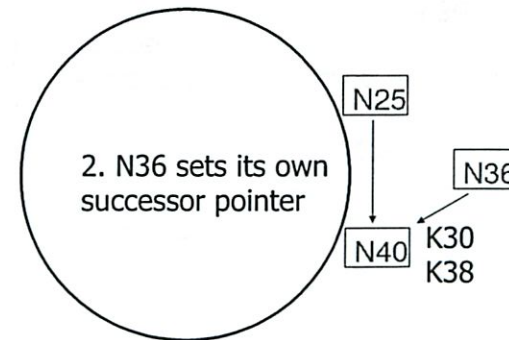
Lookups take $O(\log(N))$ hops



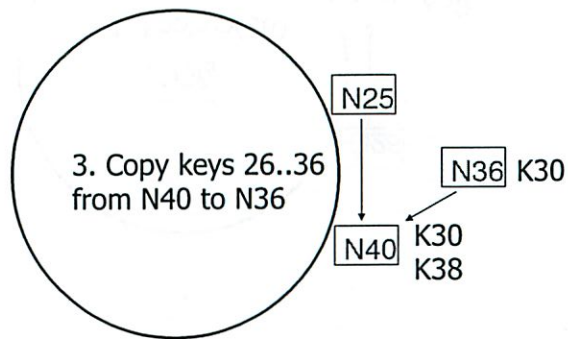
Joining: linked list insert



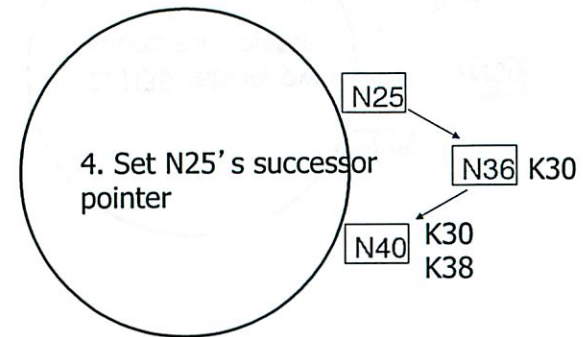
Join (2)



Join (3)

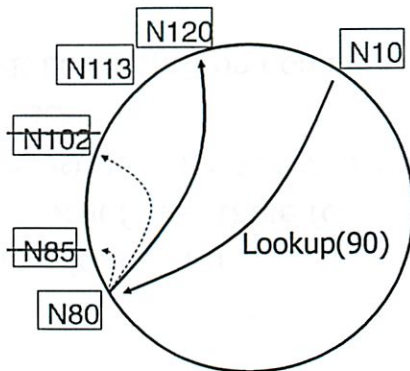


Join (4)



Update finger pointers in the background
Correct successors produce correct lookups

Failures might cause incorrect lookup



N80 doesn't know correct successor, so incorrect lookup

Solution: successor lists

- Each node knows r immediate successors
- After failure, will know first live successor
- Correct successors guarantee correct lookups
- Guarantee is with some probability

Choosing the successor list length

- Assume 1/2 of nodes fail
- $P(\text{successor list all dead}) = (1/2)^r$
 - I.e. $P(\text{this node breaks the Chord ring})$
 - Depends on independent failure
- $P(\text{no broken nodes}) = (1 - (1/2)^r)^N$
 - $r = 2\log(N)$ makes prob. = $1 - 1/N$

Other design issues

- Concurrent joins
- Locality
- Heterogeneous node
- Dishonest nodes
- ...

Lookup with fault tolerance

Lookup(my-id, key-id)

look in local finger table and successor-list
for highest node n s.t. $\text{my-id} < n < \text{key-id}$
if n exists

call Lookup(id) on node n // next hop

if call failed,

remove n from finger table

return Lookup(my-id, key-id)

else return my successor // done

Summary

- Peer-to-peer: server-less systems
 - Example: bittorrent
- Peer-to-peer lookup
 - Example: Chord

3/19

```
# -*- mode: org -*-
#+STARTUP: indent
```

6.033 2011 Lecture 12: peer-to-peer systems

* Today:

Peer-to-peer systems
hard problem: lookup
Overlay network

* Layer network model
E2E (e.g., TCP)
Network (e.g., IP)
Link (e.g., ethernet)

* Application using these stack have been client/server
Server in Machine room: well maintained, centrally located, perhaps replicated
Examples: X, DNS, master in MapReduce

* What is wrong with centralized infrastructure?
Centralized point of failure
High management costs if one org has to host millions of files, conversations, etc.
Machines owned perhaps by you and me: no obvious central authority.

* Goal: peer-to-peer system (serverless, or every client is a server)
How do you track nodes and objects in the system?
How do you find other nodes in the system (efficiently)?
How should data be split up between nodes?
How to prevent data from being lost? How to keep it available?
How to provide consistency?
How to provide security? anonymity?

* Example: bittorrent
Usages: static bulk content (Songs and videos, Linux distributions)
Other examples: Skype, etc.

** Usage model: cooperative
user downloads file from someone using simple user interface
while downloading, bittorrent serves file also to others
bittorrent keeps running for a little while after download completes

** Goal: get file out to many users quickly
Encourage everyone to upload file

** Challenges:
Tracking which peer has what
Handling high churn rates
Download rate proportional to upload rate

** Approach:
Publisher a .torrent file on a Web server (e.g., suprnova.org)
URL of tracker
file name, length
SHA1s of data blocks (64-512Kbyte)
Tracker
Organizes a swarm of peers (who has what block?)
Seed posts the URL for .torrent with tracker
Seed must have complete copy of file
Every peer that is online and has copy of a file becomes a seed
Peer asks tracker for list of peers to download from
Tracker returns list with random selection of peers
Peers contact peers to learn what parts of the file they have etc.

Download from other peers

** Transport

Peers pipeline on top of TCP

divide a block further in 16Kbyte subpieces

keep typically 5 requests in flight (to different peers)

** Which pieces to download

strict?

rarest first?

ensures that every piece is widely available

also helps with the seed and bootstrapping rapidly

won't retrieve the same piece multiple times from the seed

random?

avoid overloading seed when starting download

if peer has no piece, get as quickly as possible a piece so that it can upload

don't use rarest because it is likely only one peer has it

--> use random, can download subpieces in parallel

parallel download of same pieces?

avoid waiting on slowest

final algorithm

random for first piece, then rarest-first, parallel for last piece

** Fairness (see paper for tomorrow)

** demo: transmission with ubuntu .torrent

use inspector to look at the square of pieces, the clients, note DHTs

** Bittorrent relies on one central component: tracker.

Can we get rid off it?

Scale to large number of torrents

* Scalable lookup:

Provide an abstract interface to store and find data

Typical DHT interface:

put(key, value)

get(key) -> value

loose guarantees about keeping data alive

For bittorrent trackers:

announce tracker: put(SHA(URL), my-ip-address)

find tracker: get(SHA(url)) -> IP address of tracker

Some DHT-based trackers exist.

Many other usages of DHTs

* Goal: peer-to-peer implementation of DHT

An overlay network

partition hash table over n nodes

not every node knows about all other n nodes

rout to find right hash table

Goals:

log(n) hops

Guarantees about load balance

* Example: Chord

** ID-space topology

Ring: All IDs are 160-bit numbers, viewed in a ring.

Everyone agrees on how the ring is divided between nodes

Just based on ID bits

** Assignment of key IDs to node IDs?

Key stored on first node whose ID is equal to or greater than key ID.

Closeness is defined as the "clockwise distance"

If node and key IDs are uniform, we get reasonable load balance.
 Node IDs can be assigned, chosen randomly, SHA-1 hash of IP address...
 Key IDs can be driven from data, or chosen by user

** Routing?

Query is at some node.
 Node needs to forward the query to a node "closer" to key.
 Simplest system: either you are the "closest" or your neighbor is closer.
 Hand-off queries in a clockwise direction until done
 Only state necessary is "successor".

```
n.find_successor (k):
  if k in (n,successor]: return successor
  else: return successor.find_successor (k)
```

** Slow but steady; how can we make this faster?

This looks like a linked list: $O(n)$
 Can we make it more like a binary search?
 Need to be able to halve the distance at each step.

** Finger table routing:

Keep track of nodes exponentially further away:
 New state: $\text{succ}(n + 2^i)$
 Many of these entries will be the same in full system: expect $O(\lg N)$

```
n.find_successor (k):
  if k in (n,successor]: return successor
  else:
    n' = closest_preceding_node (k)
    return n'.find_successor (k)
```

Maybe node 8's looks like this:

```
1: 14
2: 14
4: 14
8: 21
16: 32
32: 42
```

** There's a complete tree rooted at every node

Starts at that node's row 0
 Threaded through other nodes' row 1, &c
 Every node acts as a root, so there's no root hotspot
 This is **better** than simply arranging the nodes in one tree

** How does a new node acquire correct tables?

General approach:
 Assume system starts out w/ correct routing tables.
 Use routing tables to help the new node find information.
 Add new node in a way that maintains correctness.
 Issues a lookup for its own key to any existing node.
 Finds new node's successor.
 Ask that node for its finger table.
 At this point the new node can forward queries correctly:
 Tweak its own finger table as necessary.

** Does routing *to* us now work?

If new node doesn't do anything,
 query will go to where it would have gone before we joined.
 I.e. to the existing node numerically closest to us.
 So, for correctness, we need to let people know that we are here.
 Each node keeps track of its current predecessor.
 When you join, tell your successor that its predecessor has changed.
 Periodically ask your successor who its predecessor is:

If that node is closer to you, switch to that guy.
Is that enough?

** Everyone must also continue to update their finger tables:
Periodically lookup your $n + 2^i$ -th key

** What about concurrent joins?
E.g. two new nodes with very close ids, might have same successor.
e.g. 44 and 46.
Both may find node 48... spiky tree!
Good news: periodic stabilization takes care of this.

** What about node failures?
Assume nodes fail w/o warning. Strictly harder than graceful departure.
Two issues:
Other nodes' routing tables refer to dead node.
Dead nodes predecessor has no successor.
If you try to route via dead node, detect timeout, treat as empty table entry.
I.e. route to numerically closer entry instead.
Repair: ask any node on same row for a copy of its corresponding entry.
Or any node on rows below.
All these share the right prefix.
For missing successor
Failed node might have been closest to key ID!
Need to know next-closest.
Maintain a list of successors: r successors.
If you expect really bad luck, maintain $O(\log N)$ successors.
We can route around failure.
The system is effectively self-correcting.

* Summary
** Peer-to-peer systems
** Bittorrent
peer-to-peer downloads
** Chord
An overlay network
Log N tables
Log N lookups

6.033: Computer Systems Engineering

Spring 2012

[Home / News](#)[Schedule](#)[Submissions](#)

[General Information](#)[Staff List](#)[Recitations](#)[TA Office Hours](#)

[Discussion / feedback](#)[FAQ](#)[Class Notes Errata](#)[Excellent Writing Examples](#)

[2011 Home](#)

Preparation for Recitation 13

Cool about incentives

Read "Do incentives build robustness in BitTorrent?" by Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. (This paper requires an MIT personal certificate for access.)

Read sections 1, 2, 3.2, 4, 5 and 7. The other sections are optional.

The BitTorrent protocol is based on the tit-for-tat strategy for the prisoner's dilemma game. You may wish to read The Triumph of the Golden Rule for an interesting introduction to both the prisoner's dilemma and tit-for-tat.

Think about the following questions:

- Why does tit-for-tat seem like a good idea for BitTorrent?
- Why is BitTorrent "better" than HTTP for content providers and users?
- Why do network providers not like BitTorrent?
- Is BitTyrant cheating?

✓plauding

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

[Top](#) // [6.033 home](#) //

Read 3/19

Do incentives build robustness in BitTorrent?

Michael Piatek* Tomas Isdal* Thomas Anderson* Arvind Krishnamurthy* Arun Venkataramani†

Abstract

A fundamental problem with many peer-to-peer systems is the tendency for users to “free ride”—to consume resources without contributing to the system. The popular file distribution tool BitTorrent was explicitly designed to address this problem, using a tit-for-tat reciprocity strategy to provide positive incentives for nodes to contribute resources to the swarm. While BitTorrent has been extremely successful, we show that its incentive mechanism is not robust to strategic clients. Through performance modeling parameterized by real world traces, we demonstrate that all peers contribute resources that do not directly improve their performance. We use these results to drive the design and implementation of *BitTyrant*, a strategic BitTorrent client that provides a median 70% performance gain for a 1 Mbit client on live Internet swarms. We further show that when applied universally, strategic clients can hurt average per-swarm performance compared to today’s BitTorrent client implementations.

1 Introduction

A fundamental problem with many peer-to-peer systems is the tendency of users to “free ride”—consume resources without contributing to the system. In early peer-to-peer systems such as Napster, the novelty factor sufficed to draw plentiful participation from peers. Subsequent peer-to-peer systems recognized and attempted to address the free riding problem; however, their fixes proved to be unsatisfactory, e.g., “incentive priorities” in Kazaa could be spoofed; currency in MojoNation was cumbersome; and the AudioGalaxy Satellite model of “always-on” clients has not been taken up. More recently, BitTorrent, a popular file distribution tool based on a *swarming* protocol, proposed a tit-for-tat (TFT) strategy aimed at incenting peers to contribute resources to the system and discouraging free riders.

The tremendous success of BitTorrent suggests that TFT is successful at inducing contributions from rational peers. Moreover, the bilateral nature of TFT allows for enforcement without a centralized trusted infrastructure. The consensus appears to be that “incentives build robustness in BitTorrent” [3, 17, 2, 11].

In this paper, we question this widely held belief. To this end, we first conduct a large measurement study of real BitTorrent swarms to understand the diversity of Bit-

Torrent clients in use today, realistic distributions of peer upload capacities, and possible avenues of strategic peer behavior in popular clients. Based on these measurements, we develop a simple model of BitTorrent to correlate upload and download rates of peers. We parametrize this model with the measured distribution of peer upload capacities and discover the presence of significant *altruism* in BitTorrent, i.e., all peers regularly make contributions to the system that do not directly improve their performance. Intrigued by this observation, we revisit the following question: *can a strategic peer game BitTorrent to significantly improve its download performance for the same level of upload contribution?*

Our primary contribution is to settle this question in the affirmative. Based on the insights gained from our model, we design and implement *BitTyrant*, a modified BitTorrent client designed to benefit strategic peers. The key idea is to carefully select peers and contribution rates so as to maximize download per unit of upload bandwidth. The strategic behavior of *BitTyrant* is executed simply through policy modifications to existing clients without any change to the BitTorrent protocol. We evaluate *BitTyrant* performance on real swarms, establishing that all peers, regardless of upload capacity, can significantly improve download performance while reducing upload contributions. For example, a client with 1 Mb/s upload capacity receives a median 70% performance gain from using *BitTyrant*.

How does use of *BitTyrant* by many peers in a swarm affect performance? We find that peers individually benefit from *BitTyrant*’s strategic behavior, irrespective of whether or not other peers are using *BitTyrant*. Peers not using *BitTyrant* can experience degraded performance due to the absence of altruistic contributions. Taken together, these results suggest that “incentives do not build robustness in BitTorrent”.

Robustness requires that performance does not degrade if peers attempt to strategically manipulate the system, a condition BitTorrent does not meet today. Although BitTorrent peers ostensibly make contributions to improve performance, we show that much of this contribution is unnecessary and can be reallocated or withheld while still improving performance for strategic users. Average download times currently depend on significant altruism from high capacity peers that, when withheld, reduces performance for all users.

In addition to our primary contribution, *BitTyrant*, our

*Dept. of Computer Science and Engineering, Univ. of Washington

†Dept. of Computer Science, Univ. of Massachusetts Amherst

they wrote
as an
altruistic
client!

efforts to measure and model altruism in BitTorrent are independently noteworthy. First, although modeling BitTorrent has seen a large body of work (see Section 6), our model is simpler and still suffices to capture the correlation between upload and download rates for real swarms. Second, existing studies recognizing altruism in BitTorrent consider small simulated settings or few swarms that poorly capture the diversity of deployed BitTorrent clients, peer capacities, churn, and network conditions. Our evaluation is more comprehensive. We use trace driven modeling to drive the design of *BitTyrant*, which we then evaluate on more than 100 popular, real world swarms as well as synthetic swarms on PlanetLab. Finally, we make *BitTyrant* available publicly as well as source code and anonymized traces gathered in our large-scale measurement study.

The remainder of this paper is organized as follows. Section 2 provides an overview of the BitTorrent protocol and our measurement data, which we use to parameterize our model. Section 3 develops a simple model illustrating the sources and extent of altruism in BitTorrent. Section 4 presents *BitTyrant*, a modified BitTorrent client for strategic peers, which we evaluate in Section 5. In Section 6, we discuss related work and conclude in Section 7.

2 BitTorrent overview

This section presents an overview of the BitTorrent protocol, its implementation parameters, and the measurement data we use to seed our model.

2.1 Protocol

BitTorrent focuses on bulk data transfer. All users in a particular swarm are interested in obtaining the same file or set of files. In order to initially connect to a swarm, peers download a metadata file, called a *torrent*, from a content provider, usually via a normal HTTP request. This metadata specifies the name and size of the file to be downloaded, as well as SHA-1 fingerprints of the data blocks (typically 64–512 KB) that comprise the content to be downloaded. These fingerprints are used to verify data integrity. The metadata file also specifies the address of a *tracker* server for the torrent, which coordinates interactions between peers participating in the swarm. Peers contact the tracker upon startup and departure as well as periodically as the download progresses, usually with a frequency of 15 minutes. The tracker maintains a list of currently active peers and delivers a random subset of these to clients, upon request.

Users in possession of the complete file, called *seeds*, redistribute small blocks to other participants in the swarm. Peers exchange blocks and control information with a set of directly connected peers we call the *local neighborhood*. This set of peers, obtained from the

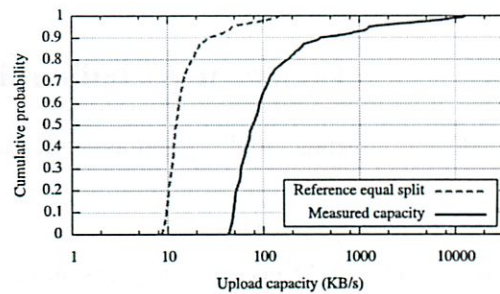


Figure 1: Cumulative distribution of raw bandwidth capacity for BitTorrent peers as well as the “equal split” capacity distribution for active set peers, assuming clients use the reference implementation of BitTorrent.

tracker, is unstructured and random, requiring no special join or recovery operations when new peers arrive or existing peers depart. The control traffic required for data exchange is minimal: each peer transmits messages indicating the data blocks they currently possess and messages signaling their interest in the blocks of other peers.

We refer to the set of peers to which a BitTorrent client is currently sending data as its *active set*. BitTorrent uses a rate-based TFT strategy to determine which peers to include in the active set. Each round, a peer sends data to unchoked peers from which it received data most rapidly in the recent past. This strategy is intended to provide positive incentives for contributing to the system and inhibit free-riding. However, clients also send data to a small number of randomly chosen peers who have not “earned” such status. Such peers are said to be optimistically unchoked. Optimistic unchokes serve to bootstrap new peers into the TFT game as well as to facilitate discovery of new, potentially better sources of data. Peers that do not send data quickly enough to earn reciprocation are removed from the active set during a TFT round and are said to be choked.

Modulo TCP effects and assuming last-hop bottleneck links, each peer provides an equal share of its available upload capacity to peers to which it is actively sending data. We refer to this rate throughout the paper as a peer’s *equal split rate*. This rate is determined by the upload capacity of a particular peer and the size of its active set. In the official reference implementation of BitTorrent, active set size is proportional to $\sqrt{\text{upload capacity}}$ (details in Appendix); although in other popular BitTorrent clients, this size is static.

2.2 Measurement

BitTorrent’s behavior depends on a large number of parameters: topology, bandwidth, block size, churn, data availability, number of directly connected peers, active TFT transfers, and number of optimistic unchokes. Furthermore, many of these parameters are a matter of pol-

← tracker?

So trades w/ good parties

individual clients (w/)

but can still have pure “leaching” clients

— so they have leaching client that improves network??

Implementation	Percentage share
Azureus	47%
BitComet	20%
μ torrent	15%
BitLord	6%
Unknown	3%
Reference	2%
Remaining	7%

Table 1: BitTorrent implementation usage as drawn from measurement data.

icy unspecified by the BitTorrent protocol itself. These policies may vary among different client implementations, and defaults may be overridden by explicit user configuration. To gain an understanding of BitTorrent's behavior and the diversity of implementations in the wild, we first conducted a measurement study of live BitTorrent swarms to ascertain client characteristics.

By making use of the opportunistic measurement techniques presented by Madhyastha et al. [14], we gather empirical measurements of BitTorrent swarms and hosts. Our measurement client connected to a large number of swarms and waited for an optimistic unchoke from each unique peer. We then estimated the upload capacity of that client using the multiQ tool [10]. Previous characterizations of end-host capacities of peer-to-peer participants were conducted by Saroiu, et al. [18]. We update these results using more recent capacity estimation tools. We observed 301,595 unique BitTorrent IP addresses over a 48 hour period during April, 2006 from 3,591 distinct ASes across 160 countries. The upload capacity distribution for typical BitTorrent peers is given in Figure 1 along with the distribution of equal split rates that would arise from peers using the reference BitTorrent implementation with no limit on upload rates.

3 Modeling altruism in BitTorrent

In this section, we examine two questions relevant to understanding how incentives impact performance in BitTorrent: how much altruism is present, and what are the sources of altruism? The first question suggests whether or not strategizing is likely to improve performance while the second informs design. Answering these questions for real world swarms is complicated by the diversity of implementations and a myriad of configuration parameters. Here, we take a restricted view and develop a model of altruism arising from our observed capacity distribution and the default parameter settings of the reference implementation of BitTorrent.

We make several assumptions to simplify our analysis and provide a conservative bound on altruism. Because our assumptions are not realistic for all swarms, our modeling results are not intended to be predictive. Rather, our results simply suggest potential sources of altruism and

the reasons they emerge in BitTorrent swarms today. We exploit these sources of altruism in the design of our real world strategic client, discussed in Section 4.

- *Representative distribution:* The CDF shown in Figure 1 is for the bandwidth capacity of observed IP addresses over many swarms. The distribution of a typical swarm may not be identical. For instance, high capacity peers tend to finish more quickly than low capacity peers, but they may also join more swarms simultaneously. If they join only a single swarm and leave shortly after completion, the relative proportion of low capacity peers would increase over the lifetime of a swarm.
- *Uniform sizing:* Peers, other than the modified client, use the active set sizing recommended by the reference BitTorrent implementation. In practice, other BitTorrent implementations are more popular (see Table 1) and have different active set sizes. As we will show, aggressive active set sizes tend to decrease altruism, and the reference implementation uses the most aggressive strategy among the popular implementations we inspected. As a result, our model provides a conservative estimate of altruism.
- *No steady state:* Active sets are comprised of peers with random draws from the overall upload capacity distribution. If churn is low, over time TFT may match peers with similar equal split rates, biasing active set draws. We argue in the next section that BitTorrent is slow to reach steady-state, particularly for high capacity peers.
- *High block availability:* Swarm performance is limited by upload capacity, i.e., peers will always be able to find interesting data to download. We find that although the reference BitTorrent implementation is designed to ensure high availability of interesting blocks, in practice, static active set sizing in some clients may degrade block availability for high capacity peers.

These assumptions allow us to model altruism in BitTorrent in terms of the upload capacity distribution only. The model is built on expressions for the probability of TFT reciprocation, expected download rate, and expected upload rate. In this section, we focus on the main insights provided by our model. The precise expressions are listed in detail in the Appendix.

3.1 Tit-for-tat matching time

Since our subsequent modeling results assume that swarms do not reach steady state, we first examine the convergence properties of the TFT strategy used to match peers of similar capacity. By default, the reference BitTorrent client optimistically unchokes two peers every 30 seconds in an attempt to explore the local neighborhood for better reciprocation pairings. Since all peers are

1 Swarm

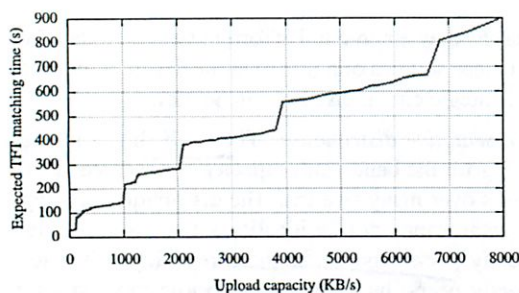


Figure 2: Assuming a peer set of infinite size, the expected time required for a new peer to discover enough peers of equal or greater equal split capacity to fill its active set.

performing this exploration concurrently, every 30 seconds a peer can expect to explore two candidate peers and be explored by two candidate peers. Since we know the equal split capacity distribution, we can express the probability of finding a peer with equal or greater equal split capacity—in a given number of 30 second rounds. Taking the expectation and multiplying it by the size of the active set gives an estimate of how long a new peer will have to wait before filling its active set with such peers.

Figure 2 shows this expected time for our observed bandwidth distribution. These results suggest that TFT as implemented does not quickly find good matches for high capacity peers, even in the absence of churn. For example, a peer with 6,400 KB/s upload capacity would transfer more than 4 GB of data before reaching steady state. In practice, convergence time is likely to be even longer. We consider a peer as being “content” with a matching once its equal split is matched or exceeded by a peer. However, one of the two peers in any matching that is not exact will be searching for alternates and switching when they are discovered, causing the other to renew its search. The long convergence time suggests a potential source of altruism: high capacity clients are forced to peer with those of low capacity while searching for better peers via optimistic unchokes.

3.2 Probability of reciprocation

A node Q sends data only to those peers in its active transfer set, reevaluated every 10 seconds. If a peer P sends data to Q at a rate fast enough to merit inclusion in Q ’s active transfer set, P will receive data during the next TFT round, and we say Q reciprocates with P .

Reciprocation from Q to P is determined by two factors: the rate at which P sends data to Q and the rates at which *other* peers send data to Q . If all other peers in Q ’s current active set send at rates greater than P , Q will not reciprocate with P .

Figure 3 gives the probability of reciprocation in terms

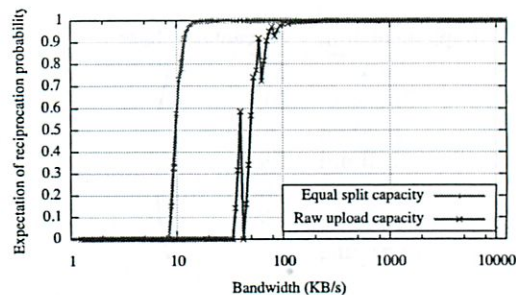


Figure 3: Reciprocation probability for a peer as a function of raw upload capacity as well as reference BitTorrent equal split bandwidth. Reciprocation probability is not strictly increasing in raw rate due to the sawtooth increase in active set size (see Table 2 in Appendix).

of both raw upload capacity and, more significantly, the equal split rate. The sharp jump in reciprocation probability suggests a potential source of altruism in BitTorrent: equal split bandwidth allocation among peers in the active set. Beyond a certain equal split rate (~ 14 KB/s in Figure 3), reciprocation is essentially assured, suggesting that further contribution may be altruistic.

3.3 Expected download rate

Each TFT round, a peer P receives data from both TFT reciprocation and optimistic unchokes. Reciprocation is possible ~~only from those peers in P ’s active set~~ and depends on P ’s upload rate, while optimistic unchokes may be received from any peer in P ’s local neighborhood, regardless of upload rate. In the reference BitTorrent client, the number of optimistic unchoke slots defaults to 2 and is rotated randomly. As each peer unchokes two peers per round, the expected number of optimistic unchokes P will receive is also two for a fixed local neighborhood size.

Figure 4 gives the expected download throughput for peers as a function of upload rate for our observed bandwidth distribution. The sub-linear growth suggests significant unfairness in BitTorrent, particularly for high capacity peers. This unfairness improves performance for the majority of low capacity peers, suggesting that high capacity peers may be able to better allocate their upload capacity to improve their own performance.

3.4 Expected upload rate

Having considered download performance, we turn next to upload contribution. Two factors can control the upload rate of a peer: data availability and capacity limit. When a peer is constrained by data availability, it does not have enough data of interest to its local neighborhood to saturate its capacity. In this case, the peer’s upload capacity is wasted and utilization suffers. Because of the dependence of upload utilization on data availability, it is crucial that a client downloads new data at a rate fast

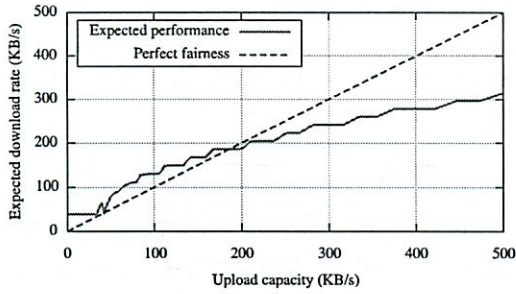


Figure 4: Expectation of download performance as a function of upload capacity. Although this represents a small portion of the spectrum of observed bandwidth capacities, $\sim 80\%$ of samples are of capacity ≤ 200 KB/s.

enough, so that the client can redistribute the downloaded data and saturate its upload capacity. We have found that indeed this is the case in the reference BitTorrent client because of the square root growth rate of its active set size.

In practice, most popular clients do not follow this dynamic strategy and instead make active set size a configurable, but static, parameter. For instance, the most popular BitTorrent client in our traces, Azureus, suggests a default active set size of four—appropriate for many cable and DSL hosts, but far lower than is required for high capacity peers. We explore the impact of active set sizing further in Section 4.1.

3.5 Modeling altruism

Given upload and download throughput, we have all the tools required to compute altruism. We consider two definitions of altruism intended to reflect two perspectives on what constitutes strategic behavior. We first consider altruism to be simply the difference between expected upload rate and download rate. Figure 5 shows altruism as a percentage of upload capacity under this definition and reflects the asymmetry of upload contribution and download rate discussed in Section 3.3. The second definition is *any* upload contribution that can be withdrawn without loss in download performance. This is shown in Figure 6.

In contrast to the original definition, Figure 6 suggests that *all* peers make altruistic contributions that could be eliminated. Sufficiently low bandwidth peers almost never earn reciprocation, while high capacity peers send much faster than the minimal rate required for reciprocation. Both of these effects can be exploited. Note that low bandwidth peers, despite not being reciprocated, still receive data in aggregate faster than they send data. This is because they receive indiscriminate optimistic unchokes from other users in spite of their low upload capacity.

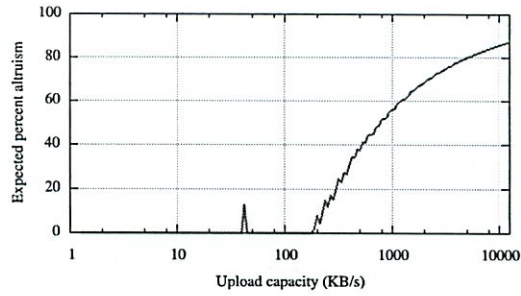


Figure 5: Expected percentage of upload capacity which is altruistic as defined by Equation 5 as a function of rate. The sawtooth increase is due to the sawtooth growth of active set sizing and equal split rates arising from integer rounding (see Table 2).

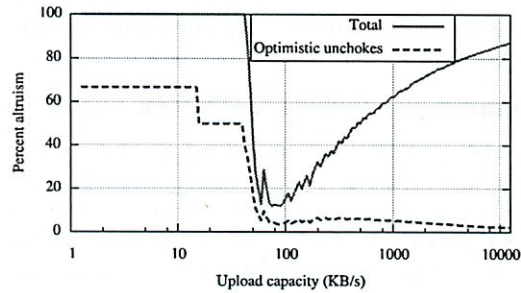


Figure 6: Expected percentage of upload capacity which is altruistic when defined as upload capacity not resulting in direct reciprocation.

3.6 Validation

Our modeling results suggest that at least part of the altruism in BitTorrent arises from the sub-linear growth of download throughput as a function of upload rate. We validate this key result using our measurement data. Each time a BitTorrent client receives a complete data block from another peer, it broadcasts a ‘have’ message indicating that it can redistribute that block to other peers. By averaging the rate of have messages over the duration our measurement client observes a peer, we can infer the peer’s download rate. Figure 7 shows this inferred download rate as a function of equal split rate, i.e., the throughput seen by the measurement client when optimistically unchoked. This data is drawn from our measurements and includes 63,482 peers.

These results indicate an even higher level of altruism than that predicted by our model (Figure 4). Note that equal split rate, the parameter of Figure 7, is a conservative lower bound on total upload capacity, shown in Figure 4, since each client sends data to many peers simultaneously. For instance, peers contributing ~ 250 KB/s to our measurement client had an observed download rate of 150 KB/s. Our model suggests that such contribution, even when split among multiple peers, should induce a

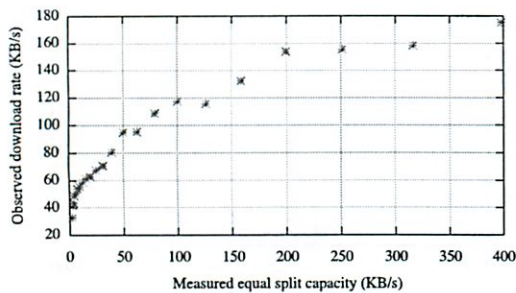


Figure 7: Measured validation of sub-linear growth in download throughput as a function of rate. Each point represents an average taken over all peers with measured equal split capacity in the intervals between points.

download rate of more than 200 KB/s. We believe this underestimate is due to more conservative active set sizes in practice than those assumed in our model.

4 Building BitTyrant: A strategic client

The modeling results of Section 3 suggest that altruism in BitTorrent serves as a kind of progressive tax. As contribution increases, performance improves, but not in direct proportion. In this section, we describe the design and implementation of *BitTyrant*, a client optimized for strategic users. We chose to base *BitTyrant* on the Azureus client in an attempt to foster adoption, as Azureus is the most popular client in our traces.

If performance for low capacity peers is disproportionately high, a strategic user can simply exploit this unfairness by masquerading as many low capacity clients to improve performance [4]. Also, by flooding the local neighborhood of high capacity peers, low capacity peers can inflate their chances of TFT reciprocation by dominating the active transfer set of a high capacity peer. In practice, these attacks are mitigated by a common client option to refuse multiple connections from a single IP address. Resourceful peers might be able to coordinate multiple IP addresses, but such an attack is beyond the capabilities of most users. We focus instead on practical strategies that can be employed by typical users.

The unfairness of BitTorrent has been noted in previous studies [2, 5, 7], many of which include protocol redesigns intended to promote fairness. However, a clean-slate redesign of the BitTorrent protocol ignores a different but important incentives question: how to get users to adopt it? As shown in Section 3, the majority of BitTorrent users benefit from its unfairness today. Designs intended to promote fairness globally at the expense of the majority of users seem unlikely to be adopted. Rather than focus on a redesign at the protocol level, we focus on BitTorrent's robustness to strategic behavior and find that strategizing can improve performance in isolation while promoting fairness at scale.

4.1 Maximizing reciprocation

The modeling results of Section 3 and the operational behavior of BitTorrent clients suggest the following three strategies to improve performance.

- Maximize reciprocation bandwidth per connection: All things being equal, a node can improve its performance by finding peers that reciprocate with high bandwidth for a low offered rate, dependent only on the other peers of the high capacity node. The reciprocation bandwidth of a peer is dependent on its upload capacity and its active set size. By discovering which peers have large reciprocation bandwidth, a client can optimize for a higher reciprocation bandwidth per connection.
- Maximize number of reciprocating peers: A client can expand its active set to maximize the number of peers that reciprocate until the marginal benefit of an additional peer is outweighed by the cost of reduced reciprocation probability from other peers.
- Deviate from equal split: On a per-connection basis, a client can lower its upload contribution to a particular peer as long as that peer continues to reciprocate. The bandwidth savings could then be reallocated to new connections, resulting in an increase in the overall reciprocation throughput.

The modeling results indicate that these strategies are likely to be effective. The largest source of altruism in our model is unnecessary contribution to peers in a node's active set. The reciprocation probability shown in Figure 3 indicates that strategically choosing equal split bandwidth can reduce contribution significantly for high capacity peers with only a marginal reduction in reciprocation probability. A peer with equal split capacity of 100 KB/s, for instance, could reduce its rate to 15 KB/s with a reduction in expected probability of reciprocation of only 1%. However, reducing from 15 KB/s to 10 KB/s would result in a decrease of roughly 40%.

The reciprocation behavior points to a performance trade-off. If the active set size is large, equal split capacity is reduced, reducing reciprocation probability. However, an additional active set connection is an additional opportunity for reciprocation. To maximize performance, a peer should increase its active set size until an additional connection would cause a reduction in reciprocation across all connections sufficient to reduce overall download performance.

If the equal split capacity distribution of the swarm is known, we can derive the active set size that maximizes the expected download rate. For our observed bandwidth distribution, Figure 8 shows the download rate as a function of the active set size for a peer with 300 KB/s upload capacity as well as the active set size that maximizes it. The graph also implicitly reflects the sensitivity of recip-

What is w/ probability isn't the protocol public?
Or some random element?

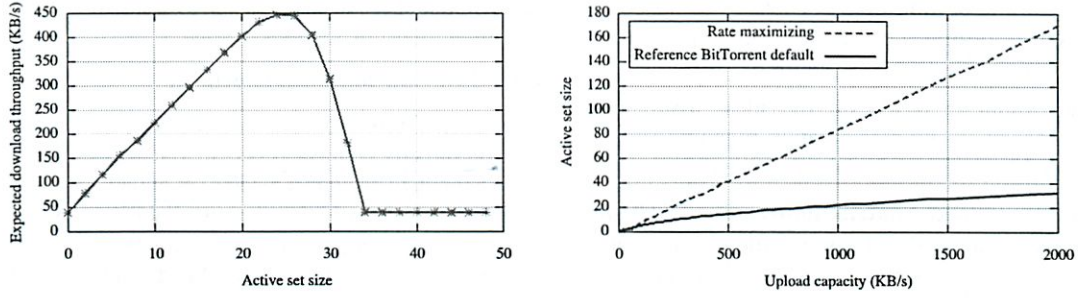


Figure 8: *Left*: The expected download performance of a client with 300 KB/s upload capacity for increasing active set size. *Right*: The performance-maximizing active set size for peers of varying rate. The strategic maximum is linear in upload capacity, while the reference implementation of BitTorrent suggests active size $\sim \sqrt{\text{rate}}$. Although several hundred peers may be required to maximize throughput, most trackers return fewer than 100 peers per request.

location probability to equal split rate.

Figure 8 is for a single strategic peer and suggests that strategic high capacity peers can benefit much more by manipulating their active set size. Our example peer with upload capacity 300 KB/s realizes a maximum download throughput of roughly 450 KB/s. However, increasing reciprocation probability via active set sizing is extremely sensitive—throughput falls off quickly after the maximum is reached. Further, it is unclear if active set sizing alone would be sufficient to maximize reciprocation in an environment with several strategic clients.

These challenges suggest that *any* a priori active set sizing function may not suffice to maximize download rate for strategic clients. Instead, they motivate the dynamic algorithm used in *BitTyrant* that adaptively modifies the size and membership of the active set and the upload bandwidth allocated to each peer (see Figure 9).

In both BitTorrent and *BitTyrant*, the set of peers that will receive data during the next TFT round is decided by the unchoke algorithm once every 10 seconds. *BitTyrant* differs from BitTorrent as it dynamically sizes its active set and varies the sending rate per connection. For each peer p , *BitTyrant* maintains estimates of the upload rate required for reciprocation, u_p , as well as the download throughput, d_p , received when p reciprocates. Peers are ranked by the ratio d_p/u_p and unchoked in order until the sum of u_p terms for unchoked peers exceeds the upload capacity of the *BitTyrant* peer.

The rationale underlying this unchoke algorithm is that the best peers are those that reciprocate most for the least number of bytes contributed to them, given accurate information regarding u_p and d_p . Implicit in the strategy are the following assumptions and characteristics:

- The strategy attempts to maximize the download rate for a given upload budget. The ranking strategy corresponds to the value-density heuristic for the knapsack problem. In practice, the download benefit (d_p) and upload cost (u_p) are not known a priori. The up-

For each peer p , maintain estimates of expected download performance d_p and upload required for reciprocation u_p .

Initialize u_p and d_p assuming the bandwidth distribution in Figure 2.

d_p is initially the expected equal split capacity of p .

u_p is initially the rate just above the step in the reciprocation probability.

Each round, rank order peers by the ratio d_p/u_p and unchoke those of top rank until the upload capacity is reached.

$$\underbrace{\frac{d_0}{u_0}, \frac{d_1}{u_1}, \frac{d_2}{u_2}, \frac{d_3}{u_3}, \frac{d_4}{u_4}, \dots}_{\text{choose } k \mid \sum_{i=0}^k u_i \leq \text{cap}}$$

At the end of each round for each unchoked peer:

If peer p does not unchoke us: $u_p \leftarrow (1 + \delta)u_p$

If peer p unchokes us: $d_p \leftarrow$ observed rate.

If peer p has unchoked us for the last r rounds:
 $u_p \leftarrow (1 - \gamma)u_p$

Figure 9: *BitTyrant* unchoke algorithm

date operation dynamically estimates these rates and, in conjunction with the ranking strategy, optimizes download rate over time.

- *BitTyrant* is designed to tap into the latent altruism in most swarms by unchoking the most altruistic peers. However, it will continue to unchoke peers until it exhausts its upload capacity even if the marginal utility is sub-linear. This potentially opens *BitTyrant* itself to being cheated, a topic we return to later.
- The strategy can be easily generalized to handle concurrent downloads from multiple swarms. A client can optimize the aggregate download rate by ordering the d_p/u_p ratios of all connections across swarms, thereby

dynamically allocating upload capacity to all peers. User-defined priorities can be implemented by using scaling weights for the d_p/u_p ratios.

The algorithm is based on the ideal assumption that peer capacities and reciprocation requirements are known. We discuss how to predict them next.

Determining upload contributions: The *BitTyrant* unchoke algorithm must estimate u_p , the upload contribution to p that induces reciprocation. We initialize u_p based on the distribution of equal split capacities seen in our measurements, and then periodically update it depending on whether p reciprocates for an offered rate. In our implementation, u_p is decreased by $\gamma = 10\%$ if the peer reciprocates for $r = 3$ rounds, and increased by $\delta = 20\%$ if the peer fails to reciprocate after being unchoked during the previous round. We use small multiplicative factors since the spread of equal split capacities is typically small in current swarms. Although a natural first choice, we do not use a binary search algorithm, which maintains upper and lower bounds for upload contributions that induce reciprocation, because peer reciprocation changes rapidly under churn and bounds on reciprocation-inducing uploads would eventually be violated.

Estimating reciprocation bandwidths: For peers that unchoke the *BitTyrant* client, d_p is simply the rate at which data was obtained from p . Note that we do not use a packet-pair based bandwidth estimation technique as suggested by Bharambe [2], but rather consider the average download rate over a TFT round. Based on our measurements, not presented here due to space limitations, we find that packet-pair based bandwidth estimates do not accurately predict peers' equal split capacities due to variability in active set sizes and end-host traffic shaping. The observed rate over a longer period is the only accurate estimate, a sentiment shared by Cohen [3].

Of course, this estimate is not available for peers that have not uploaded any data to the *BitTyrant* client. In such cases, *BitTyrant* approximates d_p for a given peer p by measuring the frequency of block announcements from p . The rate at which new blocks arrive at p provides an estimate of p 's download rate, which we use as an estimate of p 's total upload capacity. We then divide the estimated capacity by the Azureus recommended active set size for that rate to estimate p 's equal split rate. This strategy is likely to overestimate the upload capacities of unobserved peers, serving to encourage their selection from the ranking of d_p/u_p ratios. At present, this preference for exploration may be advantageous due to the high end skew in altruism. Discovering high end peers is rewarding: between the 95th and 98th percentiles, reciprocation throughput doubles. Of course, this strategy may open *BitTyrant* itself to exploitation, e.g., if a peer

rapidly announces false blocks. We discuss how to make *BitTyrant* robust in Sections 4.3 and 5.

4.2 Sizing the local neighborhood

Existing BitTorrent clients maintain a pool of typically 50–100 directly connected peers. The set is sized to be large enough to provide a diverse set of data so peers can exchange blocks without data availability constraints. However, the modeling results of Section 4.1 suggest that these typical local neighborhood sizes will not be large enough to maximize performance for high capacity peers, which may need an active set size of several hundred peers to maximize download throughput. Maintaining a larger local neighborhood also increases the number of optimistic unchokes received.

To increase the local neighborhood size in *BitTyrant*, we rely on existing BitTorrent protocol mechanisms and third party extensions implemented by Azureus. We request as many peers as possible from the centralized tracker at the maximum allowed frequency. Recently, the BitTorrent protocol has incorporated a DHT-based distributed tracker that provides peer information and is indexed by a hash of the torrent. We have increased the query rate of this as well. Finally, the Azureus implementation includes a BitTorrent protocol extension for gossip among peers. Unfortunately, the protocol extension is push-based; it allows for a client to gossip to its peers the identity of its other peers but cannot induce those peers to gossip in return. As a result, we cannot exploit the gossip mechanism to extract extra peers.

A concern when increasing the size of the local neighborhood is the corresponding increase in protocol overhead. Peers need to exchange block availability information, messages indicating interest in blocks, and peer lists. Fortunately, the overhead imposed by maintaining additional connections is modest. In comparisons of *BitTyrant* and the existing Azureus client described in Section 5, we find that average protocol overhead as a percentage of total file data received increases from 0.9% to 1.9%. This suggests that scaling the local neighborhood size does not impose a significant overhead on *BitTyrant*.

4.3 Additional cheating strategies

We now discuss more strategies to improve download performance. We do not implement these in *BitTyrant* as they can be thwarted by simple fixes to clients. We mention them here for completeness.

Exploiting optimistic unchokes: The reference BitTorrent client optimistically unchokes peers randomly. Azureus, on the other hand, makes a weighted random choice that takes into account the number of bytes exchanged with a peer. If a peer has built up a deficit in the number of traded bytes, it is less likely to be picked for optimistic unchokes. In BitTorrent today, we observe

that high capacity peers are likely to have trading deficits with most peers. A cheating client can exploit this by disconnecting and reconnecting with a different client identifier, thereby wiping out the past history and increasing its chances of receiving optimistic unchokes, particularly from high capacity peers. This exploit becomes ineffective if clients maintain the IP addresses for all peers encountered during the download and keep peer statistics across disconnections.

Downloading from seeds: Early versions of BitTorrent clients used a seeding algorithm wherein seeds upload to peers that are the fastest downloaders, an algorithm that is prone to exploitation by fast peers or clients that falsify download rate by emitting ‘have’ messages. More recent versions use a seeding algorithm that performs unchokes randomly, spreading data in a uniform manner that is more robust to manipulation.

Falsifying block availability: A client would prefer to unchoke those peers that have blocks that it needs. Thus, peers can appear to be more attractive by falsifying block announcements to increase the chances of being unchoked. In practice, this exploit is not very effective. First, a client is likely to consider most of its peers interesting given the large number of blocks in a typical swarm. Second, false announcements could lead to only short-term benefit as a client is unlikely to continue transferring once the cheating peer does not satisfy issued block requests.

5 Evaluation

To evaluate *BitTyrant*, we explore the performance improvement possible for a single strategic peer in synthetic and current real world swarms as well as the behavior of *BitTyrant* when used by all participants in synthetic swarms.

Evaluating altruism in BitTorrent experimentally and at scale is challenging. Traditional wide-area testbeds such as PlanetLab do not exhibit the highly skewed bandwidth distribution we observe in our measurements, a crucial factor in determining the amount of altruism. Alternatively, fully configurable local network testbeds such as Emulab are limited in scale and do not incorporate the myriad of performance events typical of operation in the wide-area. Further, BitTorrent implementations are diverse, as shown in Table 1.

To address these issues, we perform two separate evaluations. First, we evaluate *BitTyrant* on real swarms drawn from popular aggregation sites to measure real world performance for a single strategic client. This provides a concrete measure of the performance gains a user can achieve today. To provide more insight into how *BitTyrant* functions, we then revisit these results on PlanetLab where we evaluate sensitivity to various upload rates

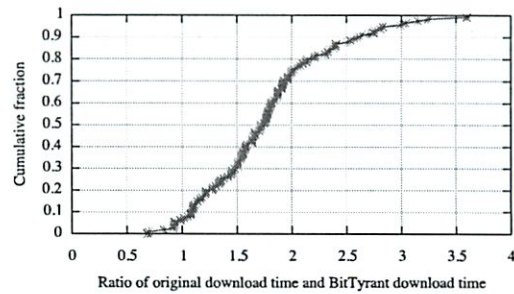


Figure 10: CDF of download performance for 114 real world swarms. Shown is the ratio between download times for an existing Azureus client and *BitTyrant*. Both clients were started simultaneously on machines at UW and were capped at 128 KB/s upload capacity.

and evaluate what would happen if *BitTyrant* is universally deployed.

5.1 Single strategic peer

To evaluate performance under the full diversity of realistic conditions, we crawled popular BitTorrent aggregation websites to find candidate swarms. We ranked these by popularity in terms of number of active participants, ignoring swarms distributing files larger than 1 GB. The resulting swarms are typically for recently released files and have sizes ranging from 300–800 peers, with some swarms having as many as 2,000 peers.

We then simultaneously joined each swarm with a *BitTyrant* client and an unmodified Azureus client with recommended default settings. We imposed a 128 KB/s upload capacity limit on each client and compared completion times. This represents a relatively well provisioned peer for which Azureus has a recommended active set size. A CDF of the ratio of original client completion time to *BitTyrant* completion time is given in Figure 10. These results demonstrate the significant, real world performance boost that users can realize by behaving strategically. The median performance gain for *BitTyrant* is a factor of 1.72 with 25% of downloads finishing at least twice as fast with *BitTyrant*. We expect relative performance gains to be even greater for clients with greater upload capacity.

These results provide insight into the performance properties of real BitTorrent swarms, some of which limit *BitTyrant*’s effectiveness. Because of the random set of peers that BitTorrent trackers return and the high skew of real world equal split capacities, *BitTyrant* cannot always improve performance. For instance, in *BitTyrant*’s worst-performing swarm, only three peers had average equal split capacities greater than 10 KB/s. In contrast, the unmodified client received eight such peers. Total download time was roughly 15 minutes, the typical minimum request interval for peers from the tracker. As a re-

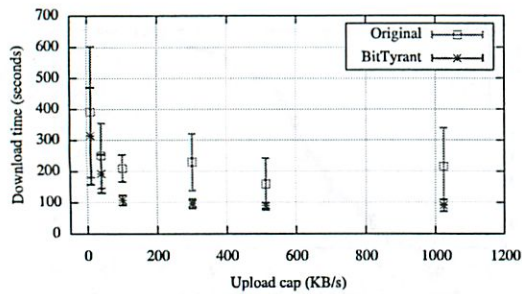


Figure 11: Download times and sample standard deviation comparing performance of a single *BitTyrant* client and an unmodified Azureus client on a synthetic PlanetLab swarm.

sult, *BitTyrant* did not recover from its initial set of comparatively poor peers. To some extent, performance can be based on luck with respect to the set of initial peers returned. More often than not, *BitTyrant* benefits from this, as it always requests a comparatively large set of peers from the tracker.

Another circumstance for which *BitTyrant* cannot significantly improve performance is a swarm whose aggregate performance is controlled by data availability rather than the upload capacity distribution. In the wild, swarms are often hamstrung by the number of peers seeding the file—i.e., those with a complete copy. If the capacity of these peers is low or if the torrent was only recently made available, there may simply not be enough available data for peers to saturate their upload capacities. In other words, if a seed with 128 KB/s capacity is providing data to a swarm of newly joined users, those peers will be able to download at a rate of at most 128 KB/s regardless of their capacity. Because many of the swarms we joined were recent, this effect may account for the 12 swarms for which download performance differed by less than 10%.

These scenarios can hinder the performance of *BitTyrant*, but they account for a small percentage of our observed swarms overall. For most real swarms today, users can realize significant performance benefits from the strategic behavior of *BitTyrant*.

Although the performance improvements gained from using *BitTyrant* in the real world are encouraging, they provide little insight into the operation of the system at scale. We next evaluate *BitTyrant* in synthetic scenarios on PlanetLab to shed light on the interplay between swarm properties, strategic behavior, and performance. Because PlanetLab does not exhibit the highly skewed bandwidth distribution observed in our traces, we rely on application level bandwidth caps to artificially constrain the bandwidth capacity of PlanetLab nodes in accordance with our observed distribution. However, because PlanetLab is often oversubscribed and shares bandwidth

equally among competing experiments, not all nodes are capable of matching the highest values from the observed distribution. To cope with this, we scaled by $1/10^{\text{th}}$ both the upload capacity draws from the distribution as well as relevant experimental parameters such as file size, initial unchoke bandwidth, and block size. This was sufficient to provide overall fidelity to our intended distribution.

Figure 11 shows the download performance for a single *BitTyrant* client as a function of rate averaged over six trials with sample standard deviation. This experiment was hosted on 350 PlanetLab nodes with bandwidth capacities drawn from our scaled distribution. Three seeds with combined capacity of 128 KB/s were located at UW serving a 5 MB file. We did not change the default seeding behavior, and varying the combined seed capacity had little impact on overall swarm performance after exceeding the average upload capacity limit. To provide synthetic churn with constant capacity, each node's *BitTyrant* client disconnected immediately upon completion and reconnected immediately.

The results of Figure 11 provide several insights into the operation of *BitTyrant*.

- *BitTyrant* does not simply improve performance, it also provides more consistent performance across multiple trials. By dynamically sizing the active set and preferentially selecting peers to optimistically unchoke, *BitTyrant* avoids the randomization present in existing TFT implementations, which causes slow convergence for high capacity peers (Section 3.1).
- There is a point of diminishing returns for high capacity peers, and *BitTyrant* can discover it. For clients with high capacity, the number of peers and their available bandwidth distribution are significant factors in determining performance. Our modeling results from Section 4.1 suggest that the highest capacity peers may require several hundred available peers to fully maximize throughput due to reciprocation. Real world swarms are rarely this large. In these circumstances, *BitTyrant* performance is consistent, allowing peers to detect and reallocate excess capacity for other uses.
- Low capacity peers can benefit from *BitTyrant*. Although the most significant performance benefit comes from intelligently sizing the active set for high capacity peers (see Figure 8), low capacity peers can still improve performance with strategic peer selection, providing them with an incentive to adopt *BitTyrant*.
- Fidelity to our specified capacity distribution is consistent across multiple trials. Comparability of experiments is often a concern on PlanetLab, but our results suggest a minimum download time determined by the capacity distribution that is consistent across trials spanning several hours. Further, the consistent performance of *BitTyrant* in comparison to unmodi-

fied Azureus suggests that the variability observed is due to policy and strategy differences and not PlanetLab variability.

5.2 Many *BitTyrant* peers

Given that all users have an individual incentive to be strategic in current swarms, we next examine the performance of *BitTyrant* when used by all peers in a swarm. We consider two types of *BitTyrant* peers: strategic and selfish. Any peer that uses the *BitTyrant* unchoking algorithm (Figure 9) is strategic. If such a peer also withholds contributing excess capacity that does not improve performance, we say it is both strategic and selfish. *BitTyrant* can operate in either mode. Selfish behavior may arise when users participate in multiple swarms, as discussed below, or simply when users want to use their upload capacity for services other than BitTorrent.

We first examine performance when all peers are strategic, i.e., use *BitTyrant* while still contributing excess capacity. Our experimental setup included 350 PlanetLab nodes with upload capacities drawn from our scaled distribution simultaneously joining a swarm distributing a 5 MB file with combined seed capacity of 128 KB/s. All peers departed immediately upon download completion. Initially, we expected overall performance to degrade since high capacity peers would finish quickly and leave, reducing capacity in the system. Surprisingly, performance improved and altruism increased. These results are summarized by the CDFs of completion times comparing *BitTyrant* and the unmodified Azureus client in Figure 12. These results are consistent with our model. In a swarm where the upload capacity distribution has significant skew, high capacity peers require many connections to maximize reciprocation. *BitTyrant* reduces bootstrapping time and results in high capacity peers having higher utilization earlier, increasing swarm capacity.

Although *BitTyrant* can improve performance, such improvement is due only to more effective use of altruistic contribution. Because *BitTyrant* can detect the point of diminishing returns for performance, these contributions can be withheld or reallocated by selfish clients. Users may choose to reallocate capacity to services other than BitTorrent or to other swarms, as most peers participate in several swarms simultaneously [7]. While all popular BitTorrent implementations support downloading from multiple swarms simultaneously, few make any attempt to intelligently allocate bandwidth among them. Those that do so typically allocate some amount of a global upload capacity to each swarm individually, which is then split equally among peers in statically sized active sets. Existing implementations cannot accurately detect when bandwidth allocated to a given swarm should be reallocated to another to improve performance.

In contrast, *BitTyrant*'s unchoking algorithm transitions naturally from single to multiple swarms. Rather than allocate bandwidth among swarms, as existing clients do, *BitTyrant* allocates bandwidth among connections, optimizing aggregate download throughput over all connections for all swarms. This allows high capacity *BitTyrant* clients to effectively participate in more swarms simultaneously, lowering per-swarm performance for low capacity peers that cannot.

To model the effect of selfish *BitTyrant* users, we repeated our PlanetLab experiment with the upload capacity of all high capacity peers capped at 100 KB/s, the point of diminishing returns observed in Figure 11. A CDF of performance under the capped distribution is shown in Figure 12. As expected, aggregate performance decreases. More interesting is the stable rate of diminishing returns *BitTyrant* identifies. As a result of the skewed bandwidth distribution, beyond a certain point peers that contribute significantly more data do not see significantly faster download rates. If peers reallocate this altruistic contribution, aggregate capacity and average performance are reduced, particularly for low capacity peers. This is reflected in comparing the performance of single clients under the scaled distribution (Figure 11) and single client performance under the scaled distribution when constrained (Figure 12). The average completion time for a low capacity peer moves from 314 to 733 seconds. Average completion time for a peer with 100 KB/s of upload capacity increases from 108 seconds to 190.

While *BitTyrant* can improve performance for a single swarm, there are several circumstances for which its use causes performance to degrade.

- If high capacity peers participate in many swarms or otherwise limit altruism, total capacity per swarm decreases. This reduction in capacity lengthens download times for all users of a single swarm regardless of contribution. Although high capacity peers will see an increase in aggregate download rate across many swarms, low capacity peers that cannot successfully compete in multiple swarms simultaneously will see a large reduction in download rates. Still, each individual peer has an incentive to be strategic as their performance improves relative to that of standard clients, even when everyone is strategic or selfish.
- New users experience a lengthy bootstrapping period. To maximize throughput, *BitTyrant* unchokes peers that send fast. New users without data are bootstrapped by the excess capacity of the system only. Bootstrapping time may be reduced by reintroducing optimistic unchokes, but it is not clear that selfish peers have any incentive to do so.
- Peering relationships are not stable. *BitTyrant* was designed to exploit the significant altruism that exists in

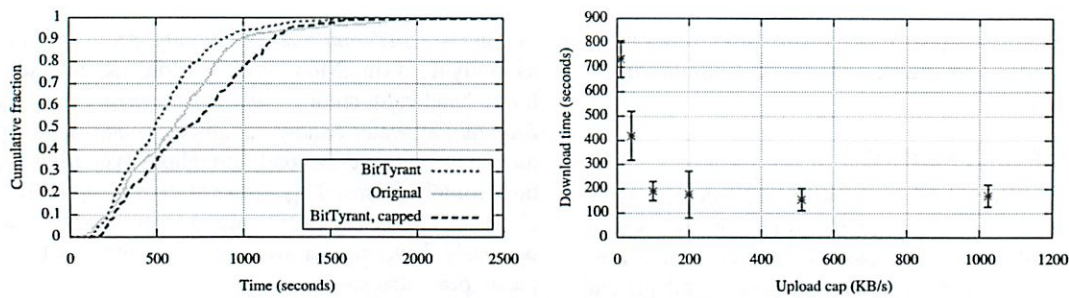


Figure 12: *Left*: CDFs of completion times for a 350 node PlanetLab experiment. *BitTyrant* and the original, unmodified client assume all users contribute all of their capacity. Capped *BitTyrant* shows performance when high capacity, selfish peers limit their contribution to the point of diminishing returns for performance. *Right*: The impact of selfish *BitTyrant* caps on performance. Download times at all bandwidth levels increase (cf. Figure 11) and high capacity peers gain little from increased contribution. Error bars give sample standard deviation over six trials.

BitTorrent swarms today. As such, it continually reduces send rates for peers that reciprocate, attempting to find the minimum rate required. Rather than attempting to ramp up send rates between high capacity peers, *BitTyrant* tends to spread available capacity among many low capacity peers, potentially causing inefficiency due to TCP effects [16].

To work around this last effect, *BitTyrant* advertises itself at connection time using the Peer ID hash. Without protocol modification, *BitTyrant* peers recognize one another and switch to a block-based TFT strategy that ramps up send rates until capacity is reached. *BitTyrant* clients choke other *BitTyrant* peers whose block request rates exceeds their send rates. By gradually increasing send and request rates to other *BitTyrant* clients, fairness is preserved while maximizing reciprocation rate with fewer connections. In this way, *BitTyrant* provides a deployment path leading to the conceptually simple strategy of block-based TFT by providing a short-term incentive for adoption by all users—even those that stand to lose from a shift to block-based reciprocation.

We do not claim that *BitTyrant* is strategyproof, even when extended with block-based TFT, and leave open for future work the question of whether further strategizing can be effective. However, a switch to block-based TFT among mutually agreeing peers would place a hard limit on altruism and limit the range of possible strategies.

6 Related work

Modeling and analysis of BitTorrent’s current incentive mechanism and its effect on performance has seen a large body of work since Cohen’s [3] seminal paper. Our effort differs from existing work in two fundamental ways. First is the *conclusion*: we refute popular wisdom that BitTorrent’s incentive mechanism makes it robust to strategic peer behavior. Second is the *methodology*: most existing studies consider small or simulated

settings that poorly capture the diversity of deployed BitTorrent clients, strategic peer behavior, peer capacities, and network conditions. In contrast, we explore BitTorrent’s strategy space with our implementation of a strategic client and evaluate it using analytical modeling, experiments under realistic network conditions, and testing in the wild.

The canonical TFT strategy was first evaluated by Axelrod [1], who showed using a competition that the strategy performs better than other submissions when there are many repeated games, persistent identities, and no collusion. Qiu and Srikant [17] specifically study BitTorrent’s rate-based TFT strategy. They show that if peers strategically limit their upload bandwidth (but split it equally) while trying to maximize download, then, under some bandwidth distributions, the system converges to a Nash equilibrium where all peers upload at their capacity. These results might lead one to believe that BitTorrent’s incentive mechanism is robust as it incentivizes users to contribute their entire upload capacities. Unfortunately, our work shows that BitTorrent fails to attain such an equilibrium for typical file sizes in swarms with realistic bandwidth distributions and churn, which *BitTyrant* exploits through strategic peer and rate selection.

Bharambe et al. [2] simulate BitTorrent using a synthetically generated distribution of peer upload capacities. They show the presence of significant altruism in BitTorrent and propose two alternate peer selection algorithms based on (i) matching peers with similar bandwidth, and (ii) enforcing TFT at the block level, a strategy also proposed by [9]. Fan et al. propose strategies for assigning rates to connections [5], which when adopted by all members of a swarm would lead to fairness and minimal altruism. The robustness of these mechanisms to strategic peer behavior is unclear. More importantly, these proposals appear to lack a convincing evolution path—a peer adopting these strategies to

day would severely hurt its download throughput as the majority of deployed conformant clients will find such a peer unattractive. In contrast, we demonstrate that *BitTyrant* can drastically reduce altruism while improving performance for a single strategic client today, incenting its adoption.

Shneidman et al. [19] identify two forms of strategic manipulation based on Sybil attacks [4] and a third based on uploading garbage data. Liogkas et al. [12] propose downloading only from seeds and also identify an exploit based on uploading garbage data. Locher et al. investigate similar techniques, i.e., ignoring rate limits of tracker requests to increase the number of available peers and connecting to as many peers as possible [13]. However, there exist straightforward fixes to minimize the impact of such “byzantine” behavior. A third exploit by Liogkas et al. involves downloading only from the fastest peers, but the strategy does not take into account the upload contribution required to induce reciprocation. In contrast, *BitTyrant* maximizes download per unit of upload bandwidth and can drastically reduce its upload contribution by varying the active set size and not sharing its upload bandwidth uniformly with active peers.

Hales and Patarin [8] argue that BitTorrent’s robustness is not so much due to its TFT mechanism, but more due to human or sociological factors that cause swarms with a high concentration of altruistic peers to be preserved over selfish ones. They further claim that releasing selfish clients into the wild may therefore not degrade performance due to the underlying natural selection. Validating this hypothesis requires building and releasing a strategic and selfish client—one of our contributions.

Massoulie and Vojnovic [15] model BitTorrent as a “coupon replication” system with a particular focus on efficiently locating the last few coupons. One of their conclusions is that altruism is not necessary for BitTorrent to be efficient. However, their study does not account for strategic behavior on the part of peers.

Other studies [2, 7, 11] have pointed out the presence of significant altruism in BitTorrent or suggest preserving it [11]. In contrast, we show that the altruism is not a consequence of BitTorrent’s incentive mechanism and can in fact be easily circumvented by a strategic client.

7 Conclusion

We have revisited the issue of incentive compatibility in BitTorrent and arrived at a surprising conclusion: although TFT discourages free riding, the bulk of BitTorrent’s performance has little to do with TFT. The dominant performance effect in practice is altruistic contribution on the part of a small minority of high capacity peers. More importantly, this altruism is not a consequence of TFT; selfish peers—even those with modest resources—can significantly reduce their contribution

and yet improve their download performance. BitTorrent works well today simply because most people use client software as-is without trying to cheat the system.

Although we have shown that selfishness can hurt swarm performance, whether or not it will do so in practice remains unclear. The public release of *BitTyrant* provides a test. Perhaps users will continue to donate their excess bandwidth, even after ensuring the maximum yield for that bandwidth. Perhaps users will behave selfishly, causing a shift to a completely different design with centrally enforced incentives. Perhaps strategic behavior will induce low bandwidth users to invest in higher bandwidth connections to compensate for their worse performance, yielding better overall swarm performance in the long run. Time will tell. These uncertainties leave us with the still open question: do incentives build robustness in BitTorrent?

The *BitTyrant* source code and distribution are publicly available at:

<http://BitTyrant.cs.washington.edu/>

Acknowledgments

We thank our shepherd, Jinyang Li, and the anonymous reviewers for their comments. This work was supported by NSF CNS-0519696 and the ARCS Foundation.

References

- [1] R. Axelrod. *The Evolution of Cooperation*. Basic Books, 1985.
- [2] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and Improving a BitTorrent Network’s Performance Mechanisms. In *Proc. of INFOCOM*, 2006.
- [3] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. of IPTPS*, 2003.
- [4] J. R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.
- [5] B. Fan, D.-M. Chiu, and J. Liu. The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design. In *Proc. of ICNP*, 2006.
- [6] GNU Scientific Library. <http://www.gnu.org/software/gsl/>.
- [7] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang. Measurements, analysis, and modeling of BitTorrent-like systems. In *Proc. of IMC*, 2005.
- [8] D. Hales and S. Patarin. How to Cheat BitTorrent and Why Nobody Does. Technical Report UBLCS 2005-12, Computer Science, University of Bologna, 2005.
- [9] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *Proc. of P2PECON*, 2005.
- [10] S. Katti, D. Katabi, C. Blake, E. Kohler, and J. Strauss. MultiQ: Automated detection of multiple bottleneck capacities along a path. In *Proc. of IMC*, 2004.
- [11] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest First and Choke Algorithms are Enough. In *Proc. of IMC*, 2006.
- [12] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent for fun (but not profit). In *Proc. of IPTPS*, 2006.

Label	Definition	Meaning
ω	2	Number of simultaneous optimistic unchokes per peer
λ	80	Local neighborhood size (directly connected peers)
$b(r)$	Figure 1	Probability of upload capacity rate r
$B(r)$	$\int_0^r b(r)dr$	Cumulative probability of a upload capacity rate r
$\text{active}(r)$	$\lfloor \sqrt{0.6r} \rfloor - \omega$	Size (in peers) of the active transfer set for upload capacity rate r
$\text{split}(r)$	$\frac{r}{\text{active}(r) + \omega}$	Per-connection upload capacity for upload capacity rate r
$s(r)$	Figure 1	Probability of an equal split rate r using mainline $\text{active}(r)$ sizing
$S(r)$	$\int_0^r s(r)dr$	Cumulative probability of an equal-split rate r

Table 2: Functions used in our model and their default settings in the official BitTorrent client.

- [13] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. of HotNets*, 2006.
- [14] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proc. of OSDI*, 2006.
- [15] L. Massoulié; and M. Vojnović. Coupon replication systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):2–13, 2005.
- [16] R. Morris. TCP behavior with many flows. In *Proc. of ICNP*, 1997.
- [17] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proc. of SIGCOMM*, 2004.
- [18] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, 2002.
- [19] J. Shneidman, D. Parkes, and L. Massoulié. Faithfulness in internet algorithms. In *Proc. of PINS*, 2004.

A Modeling notes

All numerical evaluation was performed with the GSL numerics package [6]. Refer to Section 3 for assumptions and Table 2 for definitions.

Upload / download: Probability of reciprocation for a peer P with upload capacity r_P from Q with r_Q :

$$p_{\text{recip}}(r_P, r_Q) = 1 - (1 - S(r_P))^{\text{active}(r_Q)} \quad (1)$$

Expected reciprocation probability for capacity r :

$$\text{recip}(r) = \int b(x) p_{\text{recip}}(r, x) dx \quad (2)$$

Expected download and upload rate for capacity r :

$$D(r) = \text{active}(r) \left[\int b(x) p_{\text{recip}}(r, x) \text{split}(x) dx \right] + \omega \left[\int b(x) \text{split}(x) dx \right] \quad (3)$$

$$U(r) = \min(r, (\text{active}(r) + \omega) D(r)) \quad (4)$$

Altruism: Altruism when defined as the difference between upload contribution and download reward

$$\text{altruism_gap}(r) = \max(0, U(r) - D(r)) \quad (5)$$

Altruism per connection when defined as upload contribution not resulting in direct reciprocation.

$$\begin{aligned} \text{altruism_conn}(r) = \\ \int \left(b(x) ((1 - p_{\text{recip}}(r, x)) \text{split}(r) + \right. \\ \left. p_{\text{recip}}(r, x) \max(0, \text{split}(r) - \text{split}(x))) \right) dx \end{aligned} \quad (6)$$

Total altruism not resulting in direct reciprocation.

$$\text{altruism}(r) = (\text{active}(r) + \omega) \text{altruism_conn}(r) \quad (7)$$

Convergence: Probability of a peer with rate r discovering matched TFT peer in n iterations:

$$c(r, n) = 1 - S(r)^{n \cdot 2\omega} \quad (8)$$

Time to populate active set with matched peers given upload capacity r . Note, $s = \text{split}(r)$, and $T = 30s$ is the period after which optimistic unchokes are switched.

$$\text{convergence_time}(r) = \quad (9)$$

$$T \cdot \text{active}(r) \left(c(s, 1) + \sum_{n=2}^{\infty} n c(s, n) \prod_{i=1}^{n-1} (1 - c(s, i)) \right)$$

Unchoke probability: The distribution of number of optimistic unchokes is binomial with success probability $\frac{\omega}{\lambda}$. Because overhead is low, $\lambda \gg \text{active}(r)$ in *BitTyrant*, we approximate $\lambda - \text{active}(r)$ by λ . The expected number of optimistic unchokes per round is ω .

$$\begin{aligned} Pr[\text{unchokes} = x] &= \binom{\lambda}{x} \left(\frac{\omega}{\lambda} \right)^x \left(1 - \frac{\omega}{\lambda} \right)^{(\lambda-x)} \quad (10) \\ \therefore E[\text{unchokes}] &= \lambda \frac{\omega}{\lambda} = \omega \end{aligned}$$

BitTyrant FAQ

FAQ

Q: Isn't BitTyrant just another leeching client?

No. BitTyrant does not change the *amount* of data uploaded, just *which peers* receive that data. Specifically, peers which upload more to you get more of your bandwidth. When all peers use the BitTyrant client *as released*, performance improves for the entire swarm. The details of this are explained further below. In our paper, we consider situations in which peers use clients which attempt to both maximize performance and conserve upload contribution, but BitTyrant, as released, attempts only to maximize performance.

Q: How is BitTyrant different from existing BitTorrent clients?

BitTyrant differs from existing clients in its selection of *which peers to unchoke* and *send rates* to unchoked peers. Suppose your upload capacity is 50 KBps. If you've unchoked 5 peers, existing clients will send each peer 10 KBps, independent of the rate each is sending to you. In contrast, BitTyrant will rank all peers by their receive / sent ratios, preferentially unchoking those peers with high ratios. For example, a peer sending data to you at 20 KBps and receiving data from you at 10 KBps will have a ratio of 2, and would be unchoked before unchoking someone uploading at 10 KBps (ratio 1). Further, BitTyrant dynamically adjusts its send rate, giving more data to peers that can and do upload quickly and reducing send rates to others.

Q: Will BitTyrant work for cable / DSL users?

Yes. Although the evaluation in our paper focuses on users with slightly higher upload capacity than is typically available from US cable / DSL providers today, BitTyrant's intelligent unchoking and rate selection still improves performance for users with less capacity. All users, regardless of capacity, benefit from using BitTyrant.

Q: Won't BitTyrant hurt overall BitTorrent performance if everyone uses it?

This is a subtle question and is treated most thoroughly in the paper. The short answer is: maybe. A big difference between BitTyrant and existing BitTorrent clients is that BitTyrant can detect when additional upload contribution is unlikely to improve performance. If a client were truly selfish, it might opt to withhold excess capacity, reducing performance for other users that would have received it. However, our current BitTyrant implementation always contributes excess capacity, even when it might not improve performance. *Our goal is to improve performance, not minimize upload contribution.*

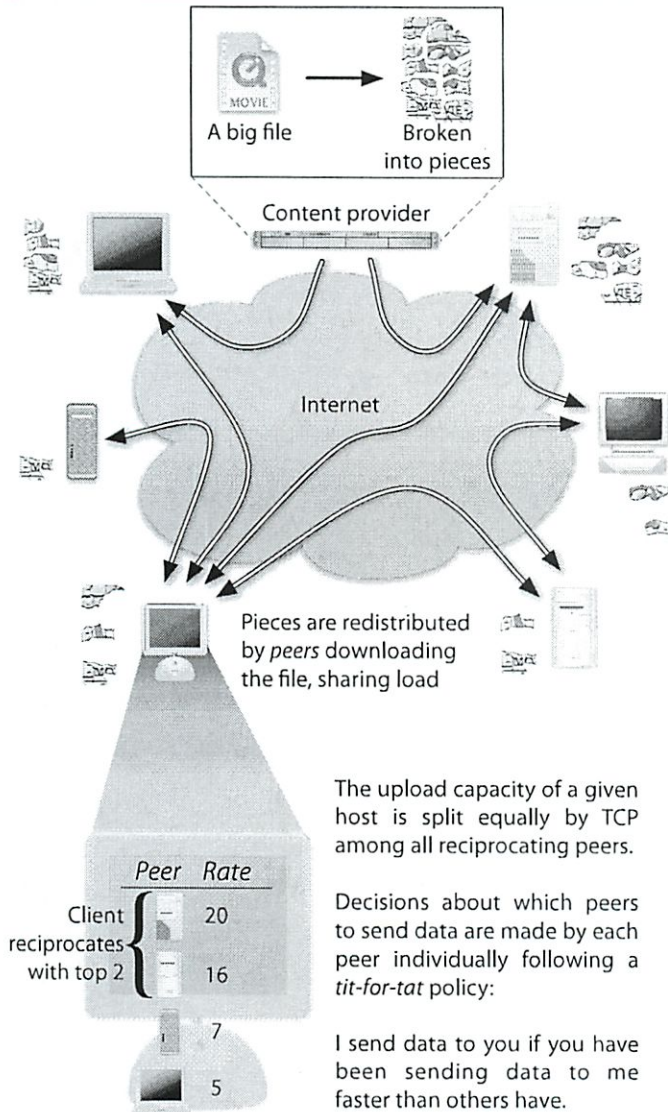
Do incentives build robustness in BitTorrent?

Michael Piatek, Tomas Isdal, Arvind Krishnamurthy, Thomas Anderson

Overview

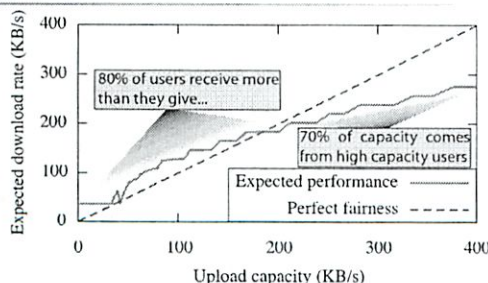
A fundamental problem with many peer-to-peer systems is the tendency of users to “free ride”—consume resources without contributing to the system. The popular file distribution tool BitTorrent was explicitly designed to address this problem, using a tit-for-tat reciprocity strategy to provide positive incentives for users to contribute resources to the system. We show that although BitTorrent has been fantastically successful, its incentive mechanism can be cheated by selfish clients.

How BitTorrent works today



Fairness

Ideally, tit-for-tat provides fairness: each person receives data as quickly as they contribute. In practice, high capacity users contribute much more than they receive.



Cheating with BitTyrant

The unfairness of BitTorrent suggests that tit-for-tat does not work as intended and might be exploited by selfish users to improve performance. We have built BitTyrant, a selfish client designed to do exactly this.

Key idea: BitTyrant dynamically chooses how many and which peers to send data. In contrast, existing BitTorrent clients send data to a fixed number of peers each tit-for-tat round, regardless of upload capacity.

Our dynamic adjustment algorithm maintains estimates of the rate at which peers will provide data, d , and the rate required to earn reciprocity, u . Using these estimates, we select the highest capacity peers and send them data at the minimum rate that will cause them to reciprocate.

Each round, rank order each peer p by the ratio d_p/u_p , and choose those of top rank until the local upload capacity is reached.

$$\frac{d_0}{u_0}, \frac{d_1}{u_1}, \frac{d_2}{u_2}, \frac{d_3}{u_3}, \frac{d_4}{u_4}, \dots$$

choose $k \mid \sum_{i=0}^k u_i \leq \text{capacity}$

At the end of each round for each unchoked peer:

If peer p does not send data: increase cost estimate, u_p .

If peer p has unchoked us for the last minute: reduce cost estimate, u_p .

In our example at left, an existing BitTorrent client might unchoke two peers based on observed received rate only. If the client had 25 KB/s of available capacity, each peer would receive data at 12.5 KB/s. In contrast, BitTyrant can determine which peers are best to exchange with and how many can be supported.

		Received rate	Required send rate	Benefit/cost ratio
Peer				
Suppose peer has capacity 25 Peer reciprocates with top 3	1	5	2	2.50
	2	20	16	1.25
	3	7	7	1.00
	4	16	16	1.00

Results

We have compared performance of BitTyrant and existing BitTorrent implementations on more than 100 real-world swarms as well as synthetic swarms on the PlanetLab testbed.

- On real swarms, BitTyrant improves download performance by 70% compared to existing BitTorrent clients. Some downloads finish more than 3 times as quickly. Regardless of capacity, using BitTyrant is in the selfish interest of every peer individually.
- However, when all peers behave selfishly, average performance degrades for all peers, even those with high capacity.

BitTyrant

From Wikipedia, the free encyclopedia

BitTyrant is a BitTorrent client modified from the Java-based Azureus 2.5 code base. BitTyrant is designed to give preference to clients uploading to it fastest and limiting slower uploaders. It is free software and cross-platform, currently available for Windows, OS X, and Linux.^[2]

BitTyrant is a result of research projects at University of Washington and University of Massachusetts Amherst, developed and supported by Professors Tom Anderson, Arvind Krishnamurthy, Arun Venkataramani and students

Michael Piatek, Jarret Falkner, and Tomas Isdal. The paper describing how it works, *Do Incentives Build Robustness in BitTorrent?*^[3], sought to challenge the common belief that BitTorrent's "must upload to download" transfer protocol prevents strategic clients from gaming the system. It won a Best Student Paper award at the 2007 Networked Systems Design and Implementation conference.

As a strategic client, it has demonstrated an average increase in download speed by 70% over a standard BitTorrent client. Non-BiTyrant leechers in the swarm may receive a decrease in download speed.^[3] Even so, if all clients are BitTyrant, high capacity peers are more effectively utilized, allowing for an overall increase in download speed. However, there is a caveat: If high capacity peers are involved in many swarms, low capacity peers lose some performance.^[3]

Contents

- 1 Strategic peer selection - an analogy
- 2 Plugins
- 3 Versions
- 4 References
- 5 External links

Strategic peer selection - an analogy

Imagine your city's central water source (the peer with data to be shared). Everyone needs water, but only a few pipes (we will suggest 10) can actually access the central source simultaneously. There are a few models

BitTyrant	
Developer(s)	University of Washington, University of Massachusetts Amherst
Stable release	1.1.1 (//en.wikipedia.org/w/index.php?title=Template:Latest_stable_software_release/BitTyrant&action=edit) (September 7, 2007 ^[1]) [±] (//en.wikipedia.org/w/index.php?title=Template:Latest_stable_software_release/BitTyrant&action=edit&preload=Template:LSR/syntax) []
Operating system	Cross-platform
Platform	Java
Type	BitTorrent client
License	GNU General Public License
Website	http://bittyrant.cs.washington.edu/

of distribution that could be adopted, two of which follow.

10 randomly selected houses might have a small hose or pipe connected to the water source. These houses, similarly, pump out 1/10 of what they receive to 10 other randomly selected houses, and so on and so forth. There is a rapid decrease in the amount of data that can be shared as one gets farther away from the central source.

A better model is to let the houses with the 10 largest pipes be connected directly to the central source. While the data is being transferred to these higher bandwidth nodes (houses), each of these in turn connects with the 10 houses that have the highest bandwidth. This accelerates the establishment of viable seeds in a torrent, and more closely corresponds to our present model, using water mains.

Very good

This example, although imperfect and somewhat exaggerated, corresponds to BitTorrent clients; the first to a standard client, and the second to BitTyrant's strategic peer selection algorithm. Clarifications of the actual algorithms used by BitTyrant follow.

When selecting which nodes have the highest bandwidth, a node uses the amount of data being received in return. Simply relying on a leecher's reported total bandwidth could easily be gamed. The seeding behavior is not modified from Azureus's standard algorithm.

*So might benefit
Community*

Plugins

Like Azureus, BitTyrant also supports the use of plugins. Plugins from Azureus such as 3D View and Safepeer can be used.

Versions

Initial release date: January 2, 2007

Version 1.1 - released January 8, 2007

Version 1.1.1 - released September 7, 2007

References

- ^a "BitTyrant" (<http://bittyrant.cs.washington.edu/>) . University of Washington (<http://www.washington.edu/>) - Computer Science & Engineering (<http://www.cs.washington.edu/>) . 2007-09-07. <http://bittyrant.cs.washington.edu/>. Retrieved 2010-01-21.
- ^a "Researchers Create Selfish BitTorrent Client" (<http://slashdot.org/article.pl?sid=07/01/03/1434259&from=rss>) . Slashdot. <http://slashdot.org/article.pl?sid=07/01/03/1434259&from=rss>. Retrieved 2007-01-03.
- ^{a b c} Michael Piatek, Tomas Isdal, Thomas Anderson, and Arvind Krishnamurthy, University of Washington; Arun Venkataramani, University of Massachusetts. Do Incentives Build Robustness in BitTorrent? Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation. 2007. <http://www.cs.washington.edu/homes/piatek/papers/BitTyrant.pdf>

External links

- BitTyrant homepage (<http://bittyrant.cs.washington.edu/>)

BitTorrent peers use tit-for-tat strategy to optimize their download speed.^[6] More specifically, most BitTorrent peers use a variant of Tit for two Tats which is called regular unchoking in BitTorrent terminology. BitTorrent peers have a limited number of upload slots to allocate to other peers. Consequently, when a peer's upload bandwidth is saturated, it will use a tit-for-tat strategy. Cooperation is achieved when upload bandwidth is exchanged for download bandwidth. Therefore, when a peer is not uploading in return to our own peer uploading, the BitTorrent program will *choke* the connection with the uncooperative peer and allocate this upload slot to a hopefully more cooperating peer. regular unchoking corresponds very strongly to always cooperating on the first move in prisoner's dilemma. Periodically, a peer will allocate an upload slot to a randomly chosen uncooperative peer (*unchoke*). This is called optimistic unchoking. This behavior allows searching for more cooperating peers and gives a second chance to previously non-cooperating peers. The optimal threshold values of this strategy are still the subject of research.

Choke = ⊗ close connection

gmilburn.ca

Essays, Projects, and Distractions of Geoff Milburn

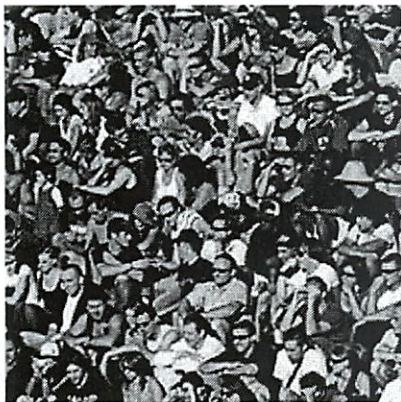
- [Subscribe](#)

Browse: [Home](#) / [Numbers & Nature](#) / Triumph of the Golden Rule

Triumph of the Golden Rule

By [Geoff](#) • February 24, 2010

Read 3/19



We live in a world with other people. Almost every decision we make involves someone else in one way or another, and we face a constant choice regarding just how much we're going to trust the person on the other side of this decision. Should we take advantage of them, go for the quick score and hope we never see them again – or should we settle for a more reasonable reward, co-operating in the hope that this peaceful relationship will continue long into the future?

We see decisions of this type everywhere, but what is less obvious is the best strategy for us to use to determine how we should act. The Golden Rule states that one should “do unto others as you would have them do unto you”. While it seems rather naive at first glance, if we run the numbers, we find something quite amazing.

A Dilemma

In order to study these types of decisions, we have to define what exactly we're talking about. Let's define just what a “dilemma” is. Let's say it has two people – and they can individually decide to work together for a shared reward, or screw the other one over and take it all for themselves. If you both decide to work together, you both get a medium-sized reward. If you decide to take advantage of someone but they trust you, you'll get a big reward (and the other person gets nothing). If you're both jerks and decide to try to take advantage of each other, you both get a tiny fraction of what you could have. Let's call these two people Alice and Bob – here's a table to make things a bit more clear.

game theory

Prisoners Dilemma

Alice cooperates

Bob cooperates Everyone wins! A medium-sized reward to both for mutual co-operation

Bob defects Poor Alice. She decided to trust Bob, who took advantage of her and got a big reward. Alice gets nothing.

Alice defects

Poor Bob. He decided to trust Alice, who screwed him and got a big reward. Bob gets nothing.

No honour among thieves... both Bob and Alice take the low road, and fight over the scraps of a small reward.

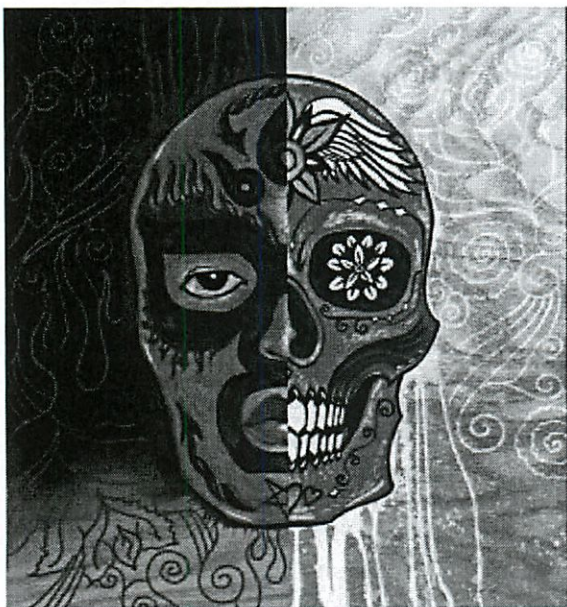
This specific order of rewards is referred to as the Prisoner's Dilemma, and was formalized and studied by Melvin Dresher and Merrill Flood in 1950 while working for the RAND Corporation.

Sale, One Day Only!

are not

Now of course the question is – if you're in this situation, what is the best thing to do? First suppose that we're never, ever going to see this other person again. This is a one time deal. Absent any moral consideration, your best option for the most profit is to attempt to take advantage of the other person and hope that they are clueless enough to let you, capitalism at its finest. You could attempt to cooperate, but that leaves you open to the other party screwing you. If each person acts in their own interest and is rational, they will attempt to one-up the other.

'if multiple rounds



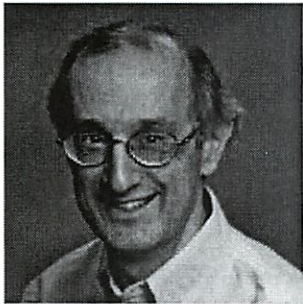
But there's just one problem – if both people act in this way, they both get much less than they would if they simply cooperated. This seems very strange, as the economic models banks and other institutions use to model human behavior assume this type of logic – the model of the rational consumer. But this leads to nearly the worst possible option if both parties take this approach.

It seems that there is no clear ideal strategy for a one time deal. Each choice leaves you open to possible losses in different ways. At this point it's easy to toss up your hands, leave logic behind, and take a moral stance. You'll cooperate because you're a good person – or you'll take advantage of the suckers because life just isn't fair.

And this appears to leave us where we are today – some good people, some bad people, and the mythical invisible hand of the market to sort them all out. But there's just one little issue. We live in a world with reputations, with friends, and with foes – there are no true “one time” deals. The world is small, and people remember.

In it for the Long Run

So instead of thinking of a single dilemma, let's think about what we should do if we get to play this game more than once. If someone screws you in the first round, you'll remember – and probably won't cooperate the next time. If you find someone who always cooperates, you can join them and work together for your mutual benefit – or decide that they're an easy mark and take them for everything they've got.



But what is the best strategy? In an attempt to figure this out, in 1980 Robert Axelrod decided to have a contest. He sent the word out, and game theorists, scientists, and mathematicians all submitted entries for a battle royale to determine which strategy was the best.

Each entry was a computer program designed with a specific strategy for playing this dilemma multiple times against other clever entries. The programs would play this simple dilemma, deciding whether to cooperate or defect against each other, for 200 rounds. Five points for a successful deception (you defect, they cooperate), three points each for mutual cooperation, one point each if you both tried to screw each other (mutual defection), and no points if you were taken advantage of (you cooperate, they defect). Each program would play every other program as well as a copy of itself, and the program with the largest total score over all the rounds would win.

So what would some very simple programs be?

ALL-C (always cooperate) is just like it sounds. Cooperation is the only way, and this program never gets tired of being an upstanding guy.

ALL-D (always defect) is the counterpoint to this, and has one singular goal. No matter what happens, always, always, always try to screw the other person over.

RAND is the lucky dunce – don't worry too much, just decide to cooperate or defect at

random.

You can predict how these strategies might do if they played against each other. Two ALL-C strategies would endlessly cooperate in a wonderful dance of mutual benefit. Two ALL-D strategies would continually fight, endlessly grinding against each other and gaining little. ALL-C pitted against ALL-D would fare about as well as a fluffy bunny in a den of wolves – eternally cooperating and hoping for reciprocation, but always getting the shaft with ALL-D profiting.

So an environment of ALL-C would be a cooperative utopia – unless a single ALL-D strategy came in, and started bleeding them dry. But an environment entirely made of ALL-D would be a wasteland – no one would have any success due to constant fighting. And the RAND strategy is literally no better than a coin flip.

Time to Think

more complicated

So what should we do? Those simple strategies don't seem to be very good at all. If we think about it however, there's a reason they do so poorly – they don't remember. No matter what the other side does, they've already made up their minds. Intelligent strategies remember previous actions of their opponents, and act accordingly. The majority of programs submitted to Axelrod's competition incorporated some sort of memory. For instance, if you can figure out you're playing against ALL-C, it's time to defect. Just like in the real world, these programs tried to figure out some concept of "reputation" that would allow them to act in the most productive manner.

And so Axelrod's competition was on. Programs from all over the world competed against each other, each trying to maximize their personal benefit. A wide variety of strategies were implemented from some of the top minds in this new field. Disk drives chattered, monitors flickered, and eventually a champion was crowned.

And the Winner Is...



When the dust settled, the winner was clear – and the victory was both surprising and inspiring. The eventual champion seemed to be a 90 lb weakling at first glance, a mere four lines of code submitted by Anatol Rapoport, a mathematical psychologist from the University of Toronto. It was called ‘Tit-for-Tat’, and it did exactly that. It started every game by cooperating – and then doing exactly what the other player did in their last turn. It cooperated with the “nice” strategies, butted heads with the “mean” strategies, and managed to come out on top ahead of far more complex approaches.

The simplest and shortest strategy won, a program that precisely enforced the Golden Rule. But what precisely made Tit-for-Tat so successful? Axelrod analyzed the results of the tournament and came up with a few principles of success.

- **Don’t get greedy.** Tit-for-Tat can never beat another strategy. But it never allows itself to take a beating, ensuring it skips the brutal losses of two “evil” strategies fighting against each other. It actively seeks out win-win situations instead of gambling for the higher payoff.
-
- **Be nice.** The single best predictor of whether a strategy would do well was if they were never the first to defect. Some tried to emulate Tit-for-Tat but with a twist – throwing in the occasional defection to up the score. It didn’t work.
-
- **Reciprocate, and forgive.** Other programs tended to cooperate with Tit-for-Tat since it consistently rewarded cooperation and punished defection. And Tit-for-Tat easily forgives – no matter how many defections it has seen, if a program decides to cooperate, it will join them and reap the rewards.
-
- **Don’t get too clever.** Tit-for-Tat is perfectly transparent, and it becomes obvious that it is very, very difficult to beat. There are no secrets, and no hypocrisy – Tit-for-Tat gets along very well with itself, unlike strategies biased toward deception.

clear
contest!

The contest attracted so much attention that a second one was organized, and this time every single entry was aware of the strategy and success of Tit-for-Tat. Sixty-three new entries arrived, all gunning for the top spot. And once again, Tit-for-Tat rose to the top. Axelrod used the results of these tournaments to develop ideas about how cooperative behaviour could evolve naturally, and eventually wrote a bestselling book called The Evolution of Cooperation. But his biggest accomplishment may be showing us that being nice does pay off – and giving us the numbers to prove it.

- Bookmark
on
- Digg
this
- Recommend
post
- on
share
- Facebook
via
- Share
Reddit
- with
tweet
- Stumblers
about
- Subscribe
to

- [Bookmark](#)
- [Comments](#)
- [Tell](#)
- [Browser](#)
- [this](#)
- [friend](#)

Categories: [Numbers & Nature](#)

Tags: [cooperation](#), [evolution](#), [Featured](#), [mathematics](#)

About the Author



Geoff

lives and works in Ontario, Canada.

Related Posts

The Golden Rule in the Wild In the previous post, we discussed the Prisoner's Dilemma and saw how a simple strategy called Tit-for-Tat enforced the Golden Rule and won a very interesting contest. But does Tit-for-Tat always come out on top? The most confounding thing about the strategy is that it can never win – at best, it can only tie [...].....

19 Responses to “Triumph of the Golden Rule”



1.

Arthur Low

February 26, 2010 at 1:36 pm | [Permalink](#) | [Reply](#)

The outcome of the contest is quite inspiring indeed.

Definitely an interesting article! Thanks for this!



2.

=== [popurls.com](#) === [popular today](#)

February 26, 2010 at 1:40 pm | [Permalink](#) | [Reply](#)

=== [popurls.com](#) === [popular today...](#)

DP1 Due Thur 5PM

Bit Tycant paper

- complicated
- but simple idea
- likely won't redo
- easier paper since DP 1

Remember Larry
has a music
reselling startup

Napster Early P2P system

Centralized server

Record labels didn't know what to do

CDs originally \$25-30

RIAA worked to keep profits high

People wanted singles

RIAA had a deal on the table

this drunk, ~~under~~ gambling uncle wanted 2x \$

Then they get sued

② RIAA was set up to prevent labels from cheating on sales counts.
Cheats end up making \$ anyway.

Napster was easy to shut down.

Originally he took points from the person who
was cheated off of
Get them to cover their papers.

Normal Download

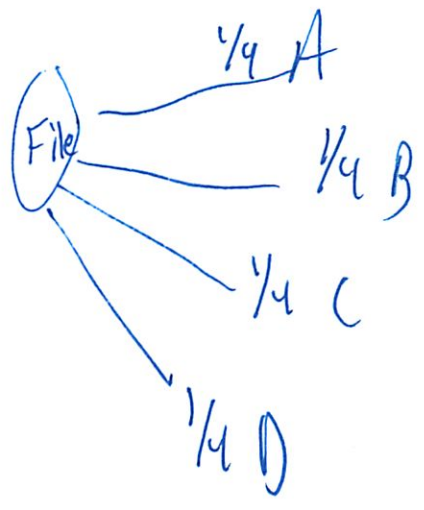
seed
(File) → Sends entire file
uploader

Networks are asymmetric
So uploads slower

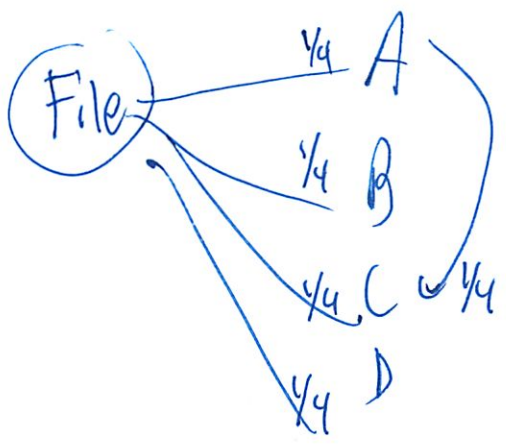
BitTorrent tries to solve problem

↳ everyone ~~wants~~ has to upload part

③



Tit for Tat - like eye for eye
match at each level



Give data to ~~the~~ my top peers
The ones who send me the most data
determined in the local client
I will upload to those people

④

He's not sharing - so I won't share w/ him!

~~He~~ Might have some peers who are better -
need to discover → opportunistic unchoking

BitTyrant takes this more steps

Let me upload the minimum amt that gets
people to still uploading to me

BitTyrant is good for ind.

BitTorrent is good for the swarm } Optimizing for
diff things

BitTyrant has no slop

- much harder to get started

5

Thomas vs RIAA

P2P Marshall - see ~~how~~ what P2P programs
are on the computer

Redigi sued for 150k per song

They sold almost no other Capital songs

Have a lot of ~~Twitter~~ Twitter followers + Client
downloads - but not uploads

Only do iTunes - selling it
"Buy"

Amazon clearer about licensing

Are you allowed to copy songs to the cloud?
- Never proved

Copy to iPod?

Backup HDD to Carbonite?

(6)

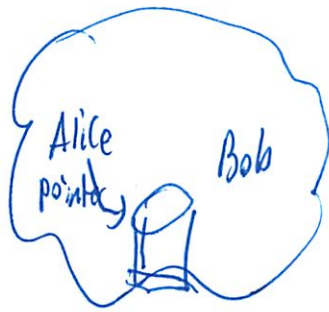
RIAA claiming song is copying

Layers said transfer songs bit by bit

So not copying

Can't have 2 copies at once

So instead a pointer



just remove pointer
and add new one



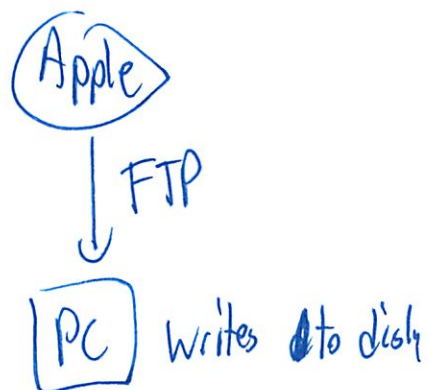
Selling copy

If you photocopy a book is used bookstore
liable,

⑦

Uploads are unencrypted
So can see is not spyware

They said the copy you have is not the original



Impose a virtual file system
Goes to bg process of Ridgi
It is ^{gets} reloaded one of the 10 downloads

Spotify doesn't pay mechanical rights

They think their tricks ~~are~~ are better than currently
legal things

Right of 1st Sale

106 - distribution

104 - resale

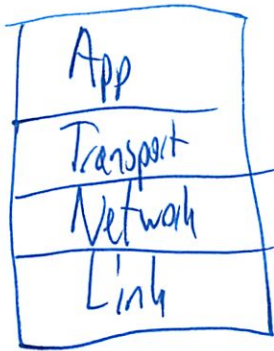
) only applies to physical goods
~~can~~ must apply same to both ways

Ge033 L13
Reliability + Congestion
Control

3/21

Final lecture on networking

The Internet Stack



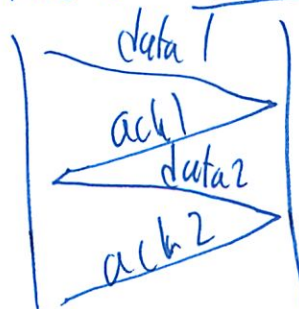
← BitTorrent does its own coding level
Treats Transport layer as link
~~network~~

Network - pretty primitive to transport layer
- No guarantees (see slide)

End to End Transport

- Reliability
at least once

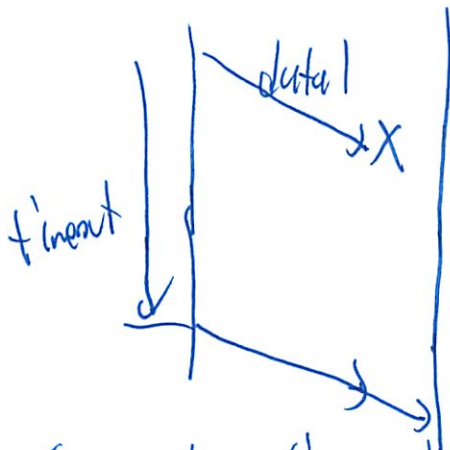
basic → lock-step



but slow!

②

How long do you wait to timeout



So set timer based on RTT
look at the ack

But RTT might be highly variable

So in TCP

Exponential Weighted Moving Avg
(see slide)

But back-step is too slow
network is idle

instead, use a window

- Could have a fixed window
- don't send > receiver's buffer

③

But still some idle time

So instead Sliding Window

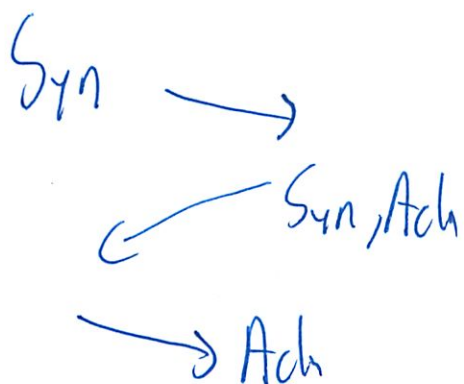
- each time we get an ack, ↑ the window
- ack and sending packets at same time
- but what should the window size be?
- (Animation of sliding window)
- If error
 - it will timeout at some point

TCP is a reliable pipe

Uses ack to adapt link capacity

Congestion

rate other server processes messages



④

Syn

~~It~~ shares

window size, packet (segment) size
time stamp - so can compare

In packets can see actual data size

TCP sometimes Slow-start

↑ window size slowly

but still increases quickly

Throughput

$$\frac{1442}{12 \text{ ms}} = 12 \text{ kbps} \text{ very slow}$$

But TCP ↑ window size quickly

then $\frac{1142 \cdot 2}{6 \text{ ms}} = 814 \text{ kb/sec}$ pretty good

Receiver window full at some point

Then $\frac{60816}{.081 \text{ sec}} = 747 \text{ kbs/sec}$

5

Packet could be received twice

- if gets lost - could be retransmitted
- and if original is then received

TCP Fast Transmit

↳ when it knows lots of packets lost
like when walk far away from base station
TCP doesn't give up as long as some packets
are received

Congestion Control

Flow control - what we saw so far

window \leq receive buffer

or else receiver would drop packets

Congestion

Don't send packets faster than
link can receive.

$\text{Tx Rate} \leq \text{Bottleneck Capacity}$

$$\text{Tx Rate} = \frac{\text{window}}{\text{RTT}}$$

(6)

So $Window \leq \min(\text{Receiver Buffer}, \text{Bottleneck Capacity} * RTT)$

Congestion window - ensures efficiency + fairness

↑ ~~the~~ cwindow slowly

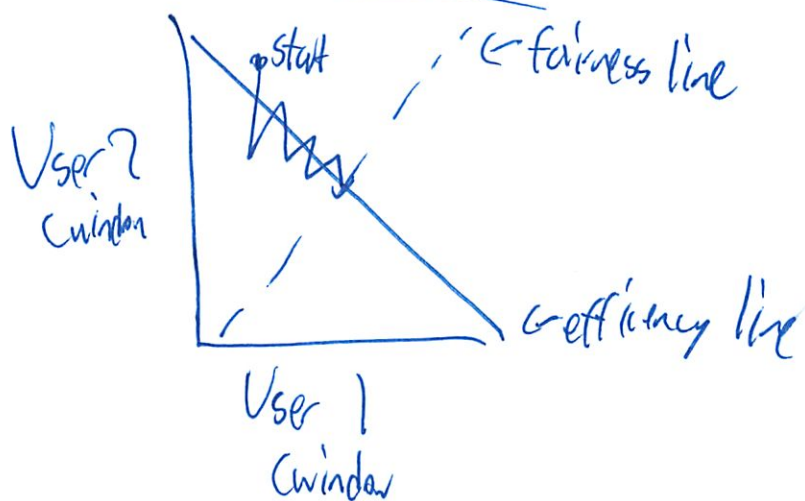
if no drops \rightarrow no congestion

if a drop \rightarrow drop cwindow quickly

$$\left. \begin{array}{l} cwindow + 1 \\ cwindow \\ \frac{cwindow}{2} \end{array} \right\}$$

- Efficiency

- Fairness



grow towards efficiency + fairness line

Even though don't know about others characteristics

Summary

$$Tx \text{ Rate} = \frac{W}{RTT}$$

$$W = \min(\text{receiver buffer}, \text{window})$$

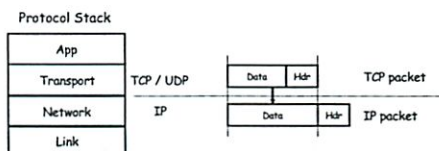
cwindow adapts

Reliability & Congestion Control

6.033 Lecture 13

Dina Katabi & Frans Kaashoek

The Internet Stack



Internet: Best Effort

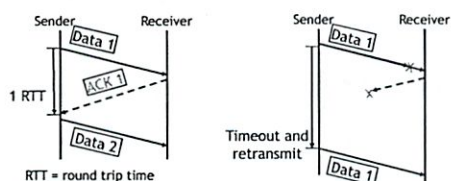
No Guarantees:

- Variable Delay (jitter)
- Variable rate
- Packet loss
- Duplicates
- Reordering
- Maximum length

E2E Transport

- ➔ • Reliability: "At Least Once Delivery"
 - Lock-step
 - Sliding Window
- Congestion Control
 - Flow Control
 - Additive Increase Multiplicative Decrease

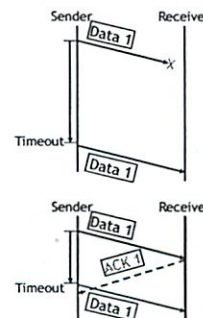
"At Least Once" (Take 1): Lock-Step



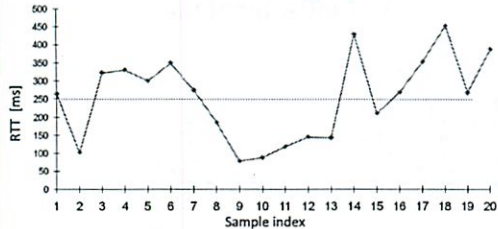
- Each data packet has a *sequence number* set by sender
- Receiver: upon receipt of packet k , sends acknowledgment (ack) for k ("I got k ")
- Sender: Upon ack k , sends $k+1$. If no ack within *timeout*, then retransmit k (until acked)

How Long to Set Timeout?

- Fixed timeouts don't work well
 - Too big → delay too long
 - Too small → unnecessary retransmission
- Solution
 - Timeout should depend on RTT
 - Sender measures the time between transmitting a packet and receiving its ack, which gives one sample of the RTT



But RTT Could Be Highly Variable



Example from a TCP connection over a wide-area wireless link
Mean RTT = 0.25 seconds; Std deviation = 0.11 seconds!

Can't set timeout to an RTT sample; need to consider variations

Calculating RTT and Timeout: (as in TCP)

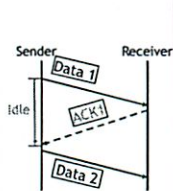
Exponentially Weighted Moving Average (EWMA)

- Estimate both the average rtt_avg and the deviation rtt_dev

• Procedure $calc_rtt(rtt_sample)$
 $rtt_avg \leftarrow a * rtt_sample + (1-a) * rtt_avg$; /* $a = 1/8$ */
 $dev \leftarrow absolute(rtt_sample - rtt_avg)$;
 $rtt_dev \leftarrow b * dev + (1-b) * rtt_dev$; /* $b = 1/4$ */

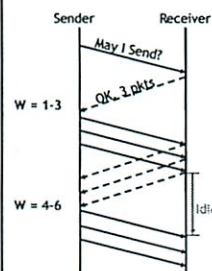
• Procedure $calc_timeout(rtt_avg, rtt_dev)$
 $Timeout \leftarrow rtt_avg + 4 * rtt_dev$

Improving Performance



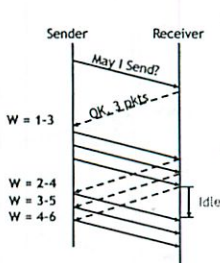
- Lock-step protocol is too slow: send, wait for ack, send, wait for ack, ...
- Throughput is just one packet per RTT
- Solution: Use a *window*
 - Keep multiple packets in the network at once
 - overlap data with acks

At Least Once (Take 2): Fixed Window



- Receiver tells the sender a window size
- Sender sends window
- Receiver acks each packet as before
- Window advances when all pkts in previous window are acked
 - E.g., packets 4-6 sent, after 1-3 ack'd
- If a packet times out \rightarrow retransmit pkt
- Still much idle time

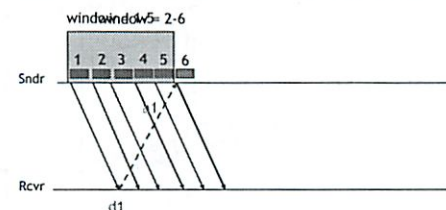
At Least Once (Take 3): Sliding Window



- Sender advances the window by 1 for each in-sequence ack it receives
 - Reduces idle periods
 - Pipelining idea!
- But what's the correct value for the window?
 - We'll revisit this question
 - First, we need to understand windows

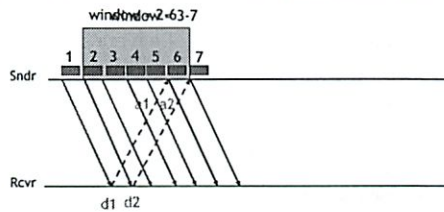
Sliding Window in Action

Example: $W = 5$; We show how the window slides with ack arrivals



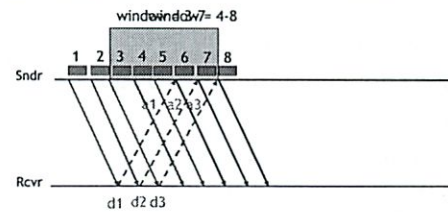
Sliding Window in Action

Example: $W = 5$; We show how the window slides with ack arrivals

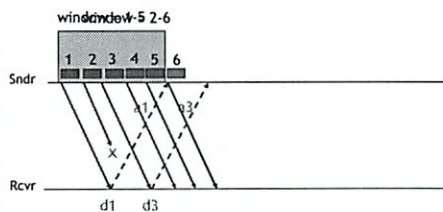


Sliding Window in Action

Example: $W = 5$; We show how the window slides with ack arrivals



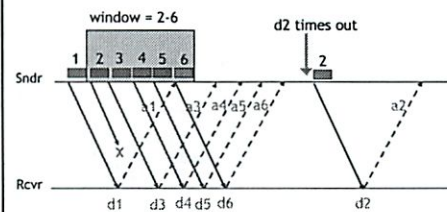
Handling Packet Loss



Sender advances the window on arrivals of in-sequence acks

→ Can't advance on a_3 's arrival

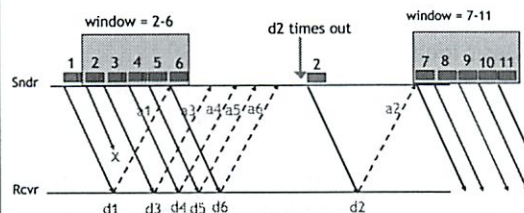
Handling Packet Loss



Sender advances the window on arrivals of in-sequence acks

→ Can't advance on a_3 's arrival

Handling Packet Loss

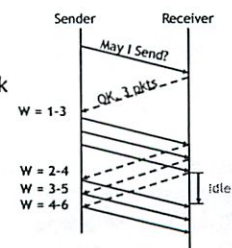


Sender advances the window on arrivals of in-sequence acks

→ Can't advance on a_3 's arrival

What is the Right Window Size?

- Window is too small
→ long Idle time
→ Underutilized Network
- Window too large
→ Congestion



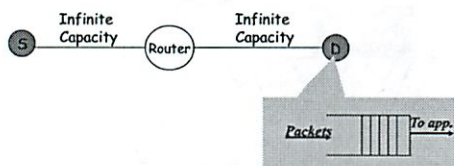
Case study: TCP

- TCP: reliable pipe to send bytes
- Uses acknowledgements to adapt to:
 - link capacity
 - rate at which server processes
 - congestion in the network
 - lost packets
- Explicit setup and tear-down

E2E Transport

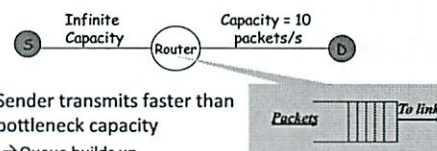
- Reliability: “At Least Once Delivery”
 - Lock-step
 - Sliding Window
- ➔ • Congestion Control
 - Flow Control
 - Additive Increase Multiplicative Decrease

Setting Window Size: Flow Control



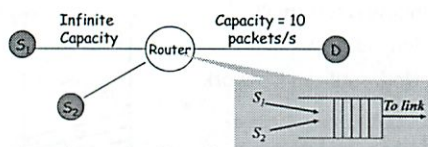
Window \leq Receiver Buffer
 – Otherwise receiver drops packets

Setting Window Size: Congestion



- Sender transmits faster than bottleneck capacity
 - Queue builds up
 - Router drops packets
- Tx Rate \leq Bottleneck Capacity
- Tx Rate = Window / RTT
- Window $\leq \min(\text{Receiver Buffer}, \text{Bottleneck_Cap} * \text{RTT})$

Setting Window Size: Congestion



Bottleneck may be shared

Window $\leq \min(\text{Receiver Buffer}, \text{cwnd})$

Congestion Control Protocol adapts the congestion window (cwnd) to ensure efficiency and fairness

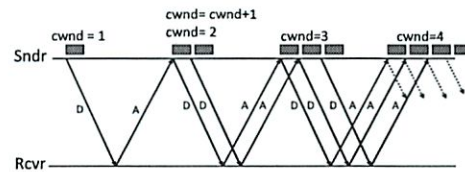
Congestion Control

- Basic Idea:
 - Increase cwnd slowly; if no drops → no congestion yet
 - If a drop occurs → decrease cwnd quickly
- Use the idea in a distributed protocol that achieves
 - Efficiency, i.e., uses the bottleneck capacity efficiently
 - Fairness, i.e., senders sharing a bottleneck get equal throughput (if they have demands)

Additive Increase Multiplicative Decrease

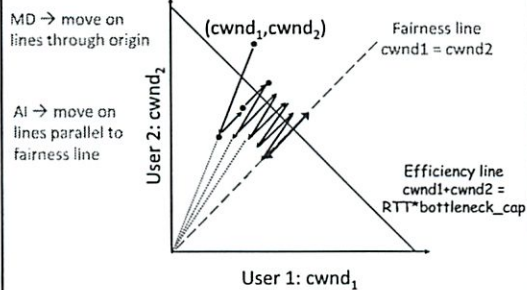
- Every RTT:
 - No drop: $\text{cwnd} = \text{cwnd} + 1$
 - A drop: $\text{cwnd} = \text{cwnd}/2$

Additive Increase



AIMD Leads to Efficiency and Fairness

Consider two users who have the same RTT



Summary of E2E Transport

- Reliability Using Sliding Window
 - Tx Rate = W / RTT
- Congestion Control
 - $W = \min(\text{Receiver_buffer}, \text{cwnd})$
 - cwnd is adapted by the congestion control protocol to ensure efficiency and fairness
 - TCP congestion control uses AIMD which provides fairness and efficiency in a distributed way

3/21

```
# -*- mode: org -*-
#+STARTUP: indent
```

Demo: wireshark trace of tcp traffic from home to amsterdam

* TCP: reliable pipe to send bytes

** Few mechanisms:

TCP adapts to link capacity, rate at which server processes packets, congestion:
all with one mechanism: the pacing of acks
TCP handles packet loss too using acks

** Semantics

3 hand-shake to set up connection (defined by src,dst ip and src, dst tcp)
server and client keep state per connection for detecting duplicate,
retransmission etc.

TCP doesn't guarantee at-most-once delivery, but in most cases it will. By
default connections hang-around for 2 min after termination, and clients pick a
random port, so it is unlikely that packets from an old connection are accepted
as new packets.

* Some observation about trace:

** connection setup: 3 packets, agree on sequence numbers, etc.

** connection starts with slow start: 1 outstanding, 2, 4, etc.

- slow start: increase the congestion window by number of packets acknowledged
first ack increase window 2
second ack increases window 4
etc. until something happens (see below)
- first ack comes back roughly in ~112 msec (what is the cable modem doing!?)
luckily we won't be sending 1448 data bytes per 112 msec (which is ~12Kbyte/s)
- after first ack sender sends 2 packets, receive ack for next packet (4345)
acknowledging 2 packets.
ack received after ~16 msec (~181 Kbyte/s)
- send 4 packets
we get ack for first packet
- mix of burst of packets, followed by receiving acks, window grows
e.g. ack for 14481 acknowledges packet send 803180-786613 usec earlier = ~16 msec
at this point we have 27513-14481 bytes in flight = 13032
so if were stable now, we would be sending at 814Kbyte/s

** TSval to compute RTT

** when sending byte 127425 connection reaches server capacity, maybe we could send faster.

- we can compute the rate at which we are sending:
byte 65161 was sent at time 66834969
it was acked at time 66916306
at that time: sender sends byte 125977
so the bytes in flight are: 125977-65161
and we are clocking at: 66834969-66916306 = -81337 (81 msec)
tcp is running at: 60816/0.081337 = 747703.99695095712898189016 bytes/sec
which make sense for uploading on a cable modem

** moved away from base station around event 9747

duplicate acks: so data got lost
several dup acks so don't wait for timeout resend: 7031633
4 dup acks, again resend 7031633
next acks says receiver send it, and retransmits two outstanding packets
my wifi link is bad so many dropped packets (i am far away from base station)
tcp retransmits and doesn't give up (client does receive some responses)

** at event 9940, i am closer to my base station

TCP is back at self-clocking itself, without drops

* Second trace: multiple TCP connections from home to am:

** First start one connection, and let it come up to speed

** At event 7143, 2 more connections

they enter slow start, reach full speed of server (tcp window is full)

** At event 9111, the combined connections are creating congestion, packet loss
many of the connection experience packet loss
connections back off aggressively:

e.g., 10672, window is 406889-367793 = 39096
e.g., 10639, window is 363449-325901 = 37548 (next ack 12 msec later, so $\sim 37548/0.012=312\text{Kb/s}$)
** They slowly speed up again
at 14712, packet loss again
at 1572, window is: 961473-931065
** and so on

Need child relations

Fix writing issues lot

Scenario - put 1 or 2 lines in

So orient myself w/ design

How do reverse order of design

So looking at file 0 - find all files
that have its parts

(I remember think about this)

Could scan for ancestors

But optional doubly linked

~~model~~ copied

Ahh - all ~~for~~ I think I remember - look
at iNode usage

(think I mentioned in it)

② So inodes have a count of log layer entries,
Log layer table has count of children

? same thing

~~and~~ would counts = ?

leaving no ...

actually I think yes

do we need both

~~inode~~

also log layer table count of ~~inode~~ links in from disk
↳ 0 means deleted

but I say 0 if not pointed to as an ancestor

So combining counts,

So lets do

inode
incoming
logs ~~from~~

log layer
incoming
links ~~from~~

~~file~~ fs

③

So is ancestor info shared?

if same ~~proving~~

inode $\leftarrow \log$
 $\leftarrow \log$

1st make 2 layer

How find ancestor?

~~And find~~ ^{visit} all. See which one has
no ancestor



optional attachment or corp

Lemailed in

Yes ^{search} ~~read~~ - prov() - ~~but this is not clear~~

3/22

(u)

So lets do incoming list of logs

① Done

I should say how call works

? Do we need to say files

I think. So also need log layer

to file pointer entries...

? What is fd?

↳ On Piazza

Adding reverse lookup log table

Have bit if deleted

Version?

↳ can lookup

Timestamp - only when actual data collected?

- not on copy + modify

- add to log level

5

Actually reverse tables could be appended
- just change location on disk

Just remove can't

Pointed to as ancestor is in the inode table
↳ just don't talk about it

Revise deleting

inode
incoming
logs

log layer
incoming
links
~~can't delete~~

links can be deleted

When no incoming links → can
delete log layer (no files share par - or
we can't report in prov)

(either it just say, no deleting --)

(6)

So when δ incoming

I still can be an ancestor

Still want to say B is based off A
even though A does not exist anymore

Originally I got the counts confused I believe

Next: Add stuff

- expand on parts

- run time analysis

- intro - design goals

- * Explain problem / purpose

I add a user field

✓

① Added compiled software

⑦

Zip/Tar

Read Piazza 265

So need to modify Zip + Tar to store
the information w/ ~~the~~ lead - prov - info

But how does it work if not portable?

Need to design a flat format
XML

Still have write prov - where it came from

Tracking down a particular part

- search the reverse for entire ~~path~~ path
- pattern match
- no "name" for parts - which makes things hard
- would need to redesign system
 - could I get away w/o it?

⑧

directories

- don't store info
- just files

Main section missing is analysis

- diff use cases
- throughput, latency, correctness
- scalability limits

I touched on all that stuff

- but give it its own section?

Conclusion

- summarize design
- problems

Ack + References - did I use any?

Word count

- any?
- textbook

⑨ Add scenarios to performance
* use cases

↑ people or things like writes

Other class: Para logical units

I think I did pretty well here---

specific business cases

Conclusion duplicate of performance
— say more its details

in future work say the problem

(This expanded nicely)

(remove dedupe in intro)

①

Add sep table for name

The Plaz Provenance File System (PPFS)

Michael Plasmeier
theplaz@mit.edu
Rudolph 10AM
March 22, 2012

Introduction

A provenance file system is a file system which stores the history and source of files edited on a local computer system. For example, when one is editing a slide deck, one might want to know from which slide deck a slide was copied from. This paper builds upon the basic file system in the early versions of Unix and introduces a provenance file system known as the *Plaz Provenance File System (PPFS)*. In particular, the PPFS introduces another layer, called the log layer, which maintains pointers to past versions and ancestors of the file.

The PPFS stores a full, verbose set of provenance information. The PPFS also stores the complete data of old versions of files, giving it many of the features of a versioning file system. The PPFS also supports seeing which files are based off a specific file. This is called reverse lookup. These characteristics make the PPFS well suited for businesses that face regulatory or legal requirements to log all changes to files. The PPFS is also well suited to businesses which frequently create derivative works from previous works. For example, a consulting company may want to know which project a slide in a slide deck was copied from.

The system aims for a simple design and it attempts to use a minimal amount of disk space for maintaining provenance information and a de minimis amount of Random Access Memory (RAM) space for the tracking of provenance information. However, it currently uses up a lot of disk space to store old versions of files. Additional disk space could be saved by de-duplication algorithms. Data is laid out so that lookups from both directions (the ancestors of a file and the children of a file) can be performed relatively quickly. As part of the operating system, the PPFS extends the usual (`read()`, `write()`) operations. For more complicated or novel functionality, new API calls are introduced.

At the moment, the PPFS operates only on one computer. It is not optimized to work over a network, nor does not track provenance information from files copied from other computers, such as web servers.

The Log Layer

The PPFS introduces a new layer into the Unix file system called the log layer. This layer is inserted between the file name layer and the inode layer, as shown in Figure 1. The file name layer is modified by redefining the inode number in the directory table to the log entry number.

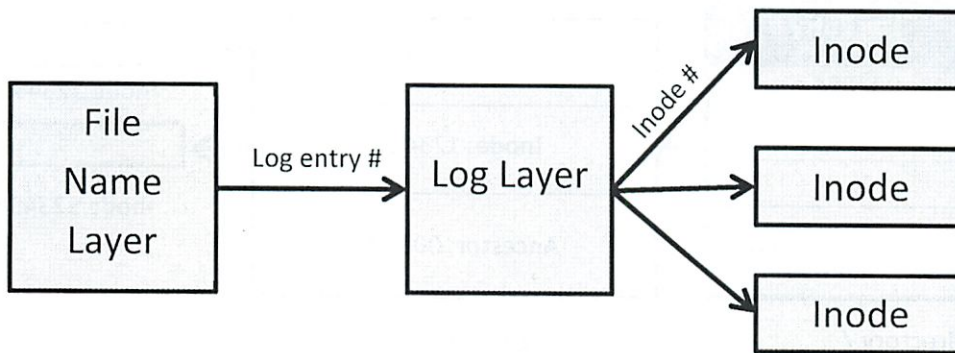


Figure 1 The Log Layer is inserted between the file name layer and the inode layer

The log layer contains a table of information about the history of each file. Each file has its own table, which is stored on disk in the same way as the inode layer. The log layer table is stored at the beginning of the disk, in a fixed position on the disk, after the inode table. The table consists of a list of log entries, each pointing to the inode number of a version of the file, as shown in Figure 2. A version is created automatically each time the file is saved. The last entry in the log layer entry contains a reference to the log layer entry that the file was created from (the ancestor). A bit in each entry designates a row as a version/inode pointer or an ancestor /log layer pointer. If a file was created from scratch (ie using touch) then the last entry in the log layer table will be 0. Although the inode stores the time the file was modified, that information would not be changed if the file was copied, so PPFS also stores that information in the log table. Additional log information, such as the current user's username and the application that made the change could also be stored here.

The number of incoming links that was stored in the inode in the original Unix file system is redefined to count the number of log layer entries pointing to the inode. The log layer table includes a count of the number of incoming links that was traditionally found in the inode. These counts are manifested in the reverse tables, described below.

Directories are ignored by the PPFS and function as usual. This may differ from certain versioning file systems.

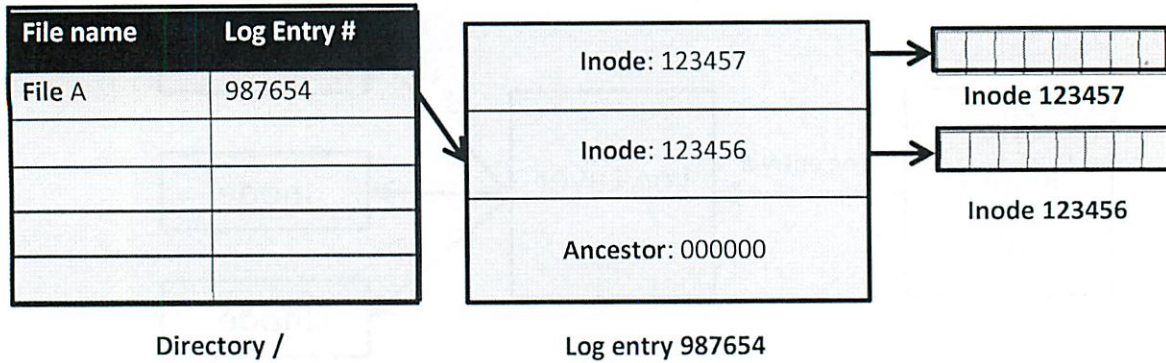


Figure 2 File A is created with content and is then edited.

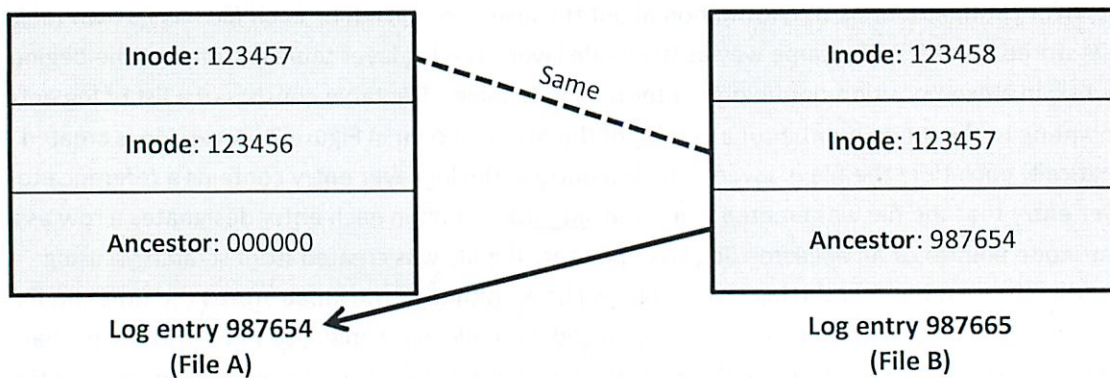


Figure 3 File A, from above, is then copied to be File B. File B is then edited. Notice that File B rev 0 shares an inode or version with file A rev 1.

When provenance information is queried (via `read_prov()`) for File B, the log entry for File B will be retrieved. Provenance information will then be recursively queried (to A in this example) until an ancestor of 0 is reached.

Reverse Lookup

One requirement of a provenance file system is to know all of the files which originated from a particular file. This information can be accessed using the `search_prov()` system call. In order to support the reverse search case, the log entry and inode tables are modified. A reverse inode table is added for each inode, which contains the list of log entry tables which point to that inode. A reverse log entry table is added to each log entry to retrieve the files names which each log entry represents.

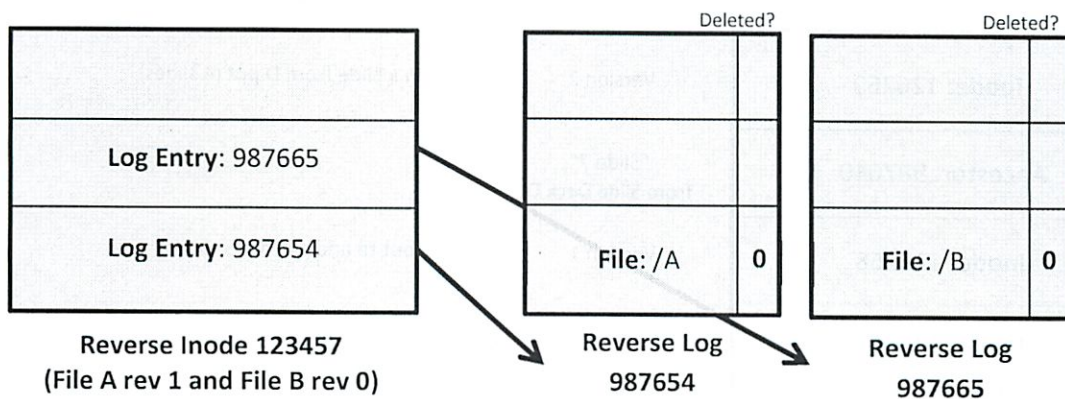


Figure 4 The Reverse Lookup Inode table for Inode 123457 shows that the File A and File B shared the same data at some point in time (i.e. one must be the ancestor of another)

In a `search_prov()` query, the system first looks up the log layer table for a particular file. For every inode mentioned in the log layer table, the system retrieves the reverse inode table. The system then retrieves the reverse log tables for each file. The system then outputs the list of files referenced. If needed, the system could also lookup the log tables themselves to find the revision number and timestamp for each file.

Parts of a File

Provenance information can be stored about the *parts* of a file (for example, the slides in a slide deck). This information is stored by having multiple ancestors in the log layer, as shown in Figure 5. In this example, the difference between Slide Deck E version 2 and version 1 came from Slide Deck D. The pointer to Slide Deck D and a name for this piece would be set by the `write_prov()` call. The data that is different would be inferred by looking at the difference between the inode before and after the ancestor entry. The name data is stored in a separate table, as seen in Figure 6.

Applications wishing to take advantage of the *provenance by parts* functionality would need to implement this API call.

Inode: 126269	Version 2	← Copy in a Slide from D.ppt (4 slides)
Ancestor: 987640	"Slide 7" from Slide Deck D	
Inode: 126268	Version 1	← Edit E.ppt to add a Slide (3 slides)
Inode: 126267	Version 0	← cp C.ppt E.ppt (2 slides)
Ancestor: 987352	Slide Deck C	

Log entry 987636
(Slide Deck E)

Figure 5 Slide Deck E was copied from Slide Deck C with 2 slides; a slide was added from scratch; and then a fourth slide was copied in from Slide Deck D (which was previously called slide 7 in Slide Deck D)

Log entry #	Name
4	"Slide 7"

Piece names 987636
(Slide Deck E)

Figure 6 Piece names for the various pieces for Slide Deck E. In this example, the 4th entry (from bottom) of the log entry for Slide Deck E were previously called "Slide 7" by the application.

Compilations of Files

Information about the source in compiled binary files can be stored in in a similar way. Multiple ancestor entries are stored at the bottom of the table between the first ancestor and the inode of the newly compiled file, as shown in Figure 7. The first entry will be 0, since the file was created new. Normal log entries will accumulate on top of this information, as before.

Inode: 126269	Compiled
Ancestor: 875884	Source G
Ancestor: 875883	Source F
Ancestor: 000000	File Created

Log entry 875855
(Binary H)

Figure 7 Binary H is compiled from Source F and Source G

File Archives

File archives present a particular challenge. File archives read information off the disk and then store it in their own proprietary format. In order to be truly portable, this requires all of the provenance information, along with all of the past versions and ancestors of a file to be stored in the file archive. This information would be retrieved using a special call, such as `read_full_provenance()`, which would store the provenance information and past versions in a flat format. This format is a XML format which mirrors the tables in the file system, as shown in Figure 8. The file would then be compressed using normal ZIP or TAR algorithms.

```
<xml schema="ppfs-portable">
  <log-entries>
    <log-entry id="875855">
      <ancestor>000000</ancestor>
      <ancestor>875883</ancestor>
      <ancestor>875884</ancestor>
      <inode>126269</inode>
      <reverse>
        <file>H</file>
      </reverse>
      //Additional metadata (i.e. name, date) removed
    </log-entry>
    <log-entry id="875883">
      <ancestor>000000</ancestor>
      <inode>126267</inode>
      <reverse>
```



```

        <file>/F</file>
    </reverse>
</log-entry>
<log-entry id="875884">
    <ancestor>000000</ancestor>
    <inode>126268</inode>
    <reverse>
        <file>/G</file>
    </reverse>
</log-entry>
</log-entries>
<inodes>
    <inode id="126269">
        <data>(Binary data)</data>
        <reverse>
            <log-entry>875855</log-entry>
        </reverse>
    </inode>
    <inode id="126268">
        <data>(Binary data)</data>
        <reverse>
            <log-entry>875883</log-entry>
        </reverse>
    </inode>
    <inode id="126267">
        <data>(Binary data)</data>
        <reverse>
            <log-entry>875884</log-entry>
        </reverse>
    </inode>
</inodes>
</xml>

```

Figure 8 The flat file XML for the scenario in Figure 6

When the file archive is extracted, the provenance information is recreated using a special `write_full_provenance()` call. The inode and table entry numbers will change, but the same structure will be created. Those that are interested in preserving the authenticity of the provenance information should disable this feature, because it allows anyone to write provenance information (including old time stamps) to disk.

Deletion and Thinning

When `unlink(filename)` is called, the filename to log entry link is removed, and the deleted bit in the reverse log table is flipped (decrementing the traditional link count in the log layer entry). When the count of incoming links in the log entry table reaches 0, the file is no longer accessible. However, the log table and versions are kept in order to preserve provenance information.

In order to save space some intermediate versions can be removed according to a thinning schedule. This schedule is user-settable, but the default values are shown in Table 1. The thinning process is accomplished by a “garbage collection”-style program. A revision will be kept if more than one log entry is present in the reverse inode table – i.e. when a file was copied and is now provenance information for a different file. When versions are thinned, the actual inode/data is removed from the disk, and all references to that version are removed from the log layer.

Days after revision created	Target number of revisions kept
< 7 days	1 / minute
> 7 days and < 30 days	1 / hour
> 30 days and < 1 year	1 / day
> 1 year	1 / week

Table 1 Intermediate revisions can be thinned after a certain amount of time after their creation. These are the default values

Performance

PPFS should be not appreciably slower when adding many files to the disk. Principally, the disk must make one additional write (the log layer table) in addition to its other writes. Generally, the non-sequential disk accesses slow a hard drive down. PPFS adds one additional non-sequential access. Thus the system should be no more than 33% slower (adding the log layer to the file system pointer, inode, and file data). Thus the system should be able to easily handle writing 10 files to disk per second. PPFS scales with the size of the disk and is linear with regard to the number of items added to disk per second. The garbage collection process is optional, and can be postponed until the system is relatively idle.

In addition, the system can quickly search for the children of a file (files that are based on that file) by using the reverse inode table. Such lookups should not depend on the number of files on the disk. PPFS scales well with regard to the number of files on disk.

For a file with many ancestors, PPFS handles reads and writes to a file the same as a file without an ancestor. Retrieving the full list of provenance information scales with the number of ancestors. It is envisioned that this will not be a large bottleneck, since the number of ancestors is envisioned to be relatively low and pulling a full list of provenance information is an infrequent operation. Caching could be added to the system to improve this time, but the additional step to update or invalidate the cache would slow the copying of files.

One of the most significant performance impacts is the time to update a file. PPFS rewrites the entire file each time it is saved, in order to maintain a version history of the file. PPFS is not optimized for large files, such as media files, and is likely unsuitable for those use cases.

PPFS uses a significant amount of disk space. PPFS is designed to provide verbosity and maintain provenance information at the expense of disk space. Thus PPFS is best suited to organizations that require comprehensive and persistent logging.

Conclusion

PPFS is a provenance file system designed to provide a comprehensive and reliable log of the history of each file. PPFS modifies the basic Unix file system to add a log layer that preserves the provenance and version history of each file on the disk. PPFS should add minimal overhead, beyond the keeping of multiple versions. PPFS should be able to easily handle lookups from both directions (the ancestors of a file and the children of a file) in a short amount of time. PPFS should scale well to the size of the disk, the number of files on disk, the number of ancestors of a file. PPFS can do additional work to reduce the disk space that old versions take up.

Implementation Issues

Modern file systems have advanced beyond the basic Unix file system that PPFS is based on. Care should be taken to maintain the current features of file systems while implementing PPFS.

Where additional API calls have been added, developers must be recruited to update their applications to support the new APIs.

Future Work

PPFS suffers from a number of limitations, which could be addressed by modifications to the system.

PPFS currently rewrites each file when it is edited, using a lot of disk space. A de-duplication algorithm which, for example, only stored the changes to a file, could save a significant amount of disk space.

PPFS currently only works on a local computer. PPFS could be expanded to work across a network. Provenance information is particularly helpful when there are multiple people working on a group of documents.

PPFS is currently designed to operate on a single disk. Because of the large amount of disk space used by PPFS, it could be expanded to work across multiple disks. For example, the RAID system allows multiple disks to be seen as one disk by a computer. This would allow users to add storage to the system as needed.

Acknowledgements

Reviewers

- Dave Custer
- Travis Grusecki

References

[1] J. Saltzer, M. F. Kaashoek, Principles of Computer System Design: An Introduction. Burlington, MA:

Morgan Kaufmann, 2009.

Word Count

2,358 words, including captions

*M.I.T. DEPARTMENT OF EECS***6.033 - Computer System Engineering****DNS Hands-On Assignment**

Hands-on 4: Internet Domain Name System

Complete the following hands-on assignment. Do the activities described, and hand in the answers to the numbered questions at the **beginning of recitation**. As usual, submit your solutions using the [online submission site](#) before recitation.

This hands-on exercise is designed to introduce you to the Internet's Domain Name System (DNS). You probably use DNS every day --- you used it to get to this page. To prepare for this assignment, please read Section 4.4 of the class textbook, titled "Case study: The Internet Domain Name System (DNS)".

Introduction

A good tool for exploring DNS is `dig`, short for Domain Information Groper. `dig` should be available on all recent Athena workstations. It should work by default, but if it does not, please try running `add watchmaker` first. If that still does not work, try an Athena Sun workstation.

Here is an example use of `dig`:

```
athena% dig slashdot.org

; <<>> DiG 9.3.1 <<>> slashdot.org
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 997
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
slashdot.org.                IN      A

;; ANSWER SECTION:
slashdot.org.                3600    IN      A      216.34.181.45      (*)

;; AUTHORITY SECTION:
slashdot.org.                86399   IN      NS      ns-2.ch3.sourceforge.com.
slashdot.org.                86399   IN      NS      ns-1.ch3.sourceforge.com.
slashdot.org.                86399   IN      NS      ns-1.sourceforge.com.

;; ADDITIONAL SECTION:
ns-1.ch3.sourceforge.com.    172800 IN      A      216.34.181.21
ns-1.sourceforge.com.        172800 IN      A      208.122.22.23
ns-2.ch3.sourceforge.com.    172800 IN      A      216.34.181.22

;; Query time: 69 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Wed Mar 11 17:32:51 2009
;; MSG SIZE rcvd: 170
```

`dig` performs a DNS lookup and prints information about the request and the response it received. If

you run `dig`, you may see results that differ from those presented here. At the bottom, we can see that the query was sent to our default server (127.0.0.1), and that it took roughly 69 msec to respond. Most of the information we are interested in is in the `ANSWER` section, marked with a (*) above. Let's examine that section more closely:

```
;; ANSWER SECTION:
slashdot.org.      3600    IN      A       216.34.181.45
      name      expire  class  type    data (IP)
```

We can see that this result is of type `A`, an address record: it is telling us that the IP address for the name "slashdot.org" is 216.34.181.45. The expiry time field "3600" indicates that this record/entry is valid for 3600 seconds (1 hour). You can ignore the "class" field; this is nearly always `IN` for Internet.

The `AUTHORITY` section contains records of type `NS`, indicating the names of DNS servers that have name records for a particular domain. Here, we can see that three DNS servers (ns-1.ch3.sourceforge.com., ns-1.sourceforge.com. and ns-2.ch3.sourceforge.com.) are responsible for answering requests for names in the slashdot.org domain.

We can ask a specific server (instead of the default) for information about a host by using the following syntax:

```
athena% dig @amsterdam.lcs.mit.edu slashdot.org

; <<>> DiG 9.3.1 <<>> @amsterdam.lcs.mit.edu slashdot.org
; (1 server found)
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1988
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;slashdot.org.                IN      A

;; ANSWER SECTION:
slashdot.org.      3600    IN      A       216.34.181.45

...[output truncated]
```

The `rd` (recursion desired) flag indicates that `dig` requested a recursive lookup, and the `ra` (recursion available) flag indicates that the server permits recursive lookups (some do not).

`dig` only prints the final result of the recursive search. You can mimic the individual steps of a recursive search by sending a request to a particular DNS server and asking for no recursion, using the `+norecurs` flag. For example, to send a non-recursive query to one of the root servers:

```
athena% dig @a.ROOT-SERVERS.NET www.slashdot.org +norecurs

; <<>> DiG 9.3.1 <<>> @a.ROOT-SERVERS.NET www.slashdot.org +norecurs
;; (1 server found)
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1888
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 6, ADDITIONAL: 12

;; QUESTION SECTION:
;www.slashdot.org.           IN      A
```

```
;; AUTHORITY SECTION:
org.          172800 IN      NS      B0.ORG.AFILIAS-NST.org.
org.          172800 IN      NS      A0.ORG.AFILIAS-NST.INFO.
org.          172800 IN      NS      A2.ORG.AFILIAS-NST.INFO.
org.          172800 IN      NS      D0.ORG.AFILIAS-NST.org.
org.          172800 IN      NS      C0.ORG.AFILIAS-NST.INFO.
org.          172800 IN      NS      B2.ORG.AFILIAS-NST.org.

;; ADDITIONAL SECTION:
A0.ORG.AFILIAS-NST.INFO. 172800 IN      A      199.19.56.1
A0.ORG.AFILIAS-NST.INFO. 172800 IN      AAAA   2001:500:e::1
A2.ORG.AFILIAS-NST.INFO. 172800 IN      A      199.249.112.1
A2.ORG.AFILIAS-NST.INFO. 172800 IN      AAAA   2001:500:40::1
B0.ORG.AFILIAS-NST.org.  172800 IN      A      199.19.54.1
B0.ORG.AFILIAS-NST.org.  172800 IN      AAAA   2001:500:c::1
B2.ORG.AFILIAS-NST.org.  172800 IN      A      199.249.120.1
B2.ORG.AFILIAS-NST.org.  172800 IN      AAAA   2001:500:48::1
C0.ORG.AFILIAS-NST.INFO. 172800 IN      A      199.19.53.1
C0.ORG.AFILIAS-NST.INFO. 172800 IN      AAAA   2001:500:b::1
D0.ORG.AFILIAS-NST.org.  172800 IN      A      199.19.57.1
D0.ORG.AFILIAS-NST.org.  172800 IN      AAAA   2001:500:f::1

;; Query time: 84 msec
;; SERVER: 198.41.0.4#53(198.41.0.4)
;; WHEN: Wed Mar 11 17:45:41 2009
;; MSG SIZE rcvd: 436
```

As you can see, the server does not know the answer and instead provides information about the servers most likely to be able to provide authoritative information. In this case, the best the root server knows is the identities of the servers for the `org.` domain.

Here are some exercises. You should submit answers *only* to the questions asked. In particular, please do not include pages of output from `dig` unless specifically requested. As usual, submit your solutions using the [online submission site](#), before recitation.

I. Getting started

- (1) Using `dig`, find the IP address for `thyme.lcs.mit.edu`. What is the IP address?
- (2) The `dig` answer for `thyme` includes a record of type `CNAME`. In the terminology of chapter 4, what does `CNAME` mean?
- (3) What is the expiration time for the `thyme` `CNAME` record?
- (4) **Note that a previous version of this hands-on had you run different commands for this question. If you already answered questions 4 and 5 with the older version, you don't need to redo this part of the assignment.**

Run these commands to find what the computer you're using gets when it looks up `"ai"` and `"ai."`.

```
dig +domain=mit.edu ai
dig +domain=mit.edu ai.
```

What are the two resulting IP addresses?

- (5) Why are the results different? Look at the man page for `dig` to see what the `+domain=` parameter does. Based on the output of the two commands, what is the difference between the DNS searches being performed for `ai` and `ai.`?

II. Understanding hierarchy

For this problem, you will go through the steps of resolving a particular hostname, mimicing a standard recursive query. Assuming it knows nothing else about a name, a DNS resolver will ask a well-known *root server*. The root servers on the Internet are in the domain `root-servers.net`. One way to get a list of them is with the command:

```
athena% dig . ns
```

- (6) Use `dig` to ask *one* of the root servers the address of `lirone.csail.mit.edu`, *without* recursion. What command do you use to do this?
- (7) It is unlikely that these servers actually know the answer so they will *refer* you to a server (or list of servers) that might know more. Go through the hierarchy from the root **without recursion**, following the referrals manually, until you have found the address of `lirone.csail.mit.edu`. What commands did you use to do this? What IP address did you find for `lirone`?

III. Understanding caching

These queries will show you how your local machine's DNS cache works.

- (8) Ask your default server for information, without recursion, about the host `www.dmoz.org`. What command did you use? Did your default server have the answer in its cache? How do you know? How long did this query take? If this information was cached, please find some other host name that is not cached and do this section with that other host.
 - (9) Now, ask your default server this same query but *with* recursion. It should return an answer for you. How long did this take?
 - (10) Finally, ask your default server again without recursion. How long does this request take? Has the cache served its purpose?
-

DNS

dig - lookup for DNS

(can ask a specific server w/ ①)

hmm dig not work for me

Is it 1600 seconds from each

↳ No I saw time goes to each time

Just by luck had a even # first time round.

Root servers

root-servers.net

or dig . ns

They gave us that command before!

↳ so edu root servers are also "fake"?

↳ well it gives you their IP!

Hands-on 4: DNS

Michael Plasmeier

1. 18.26.0.122
2. CNAME means go ask this server about the DNS records for the given domain.
3. 1627 seconds from the time the info was retrieved. So for the query run at Thu Mar 29 00:16:00 2012 then 1627 seconds is today at 00:43:07.
If you rerun the query, you notice that the number of seconds decreases.
4. ai is 128.52.32.80
ai. is 209.59.119.34
5. The domain parameter appears to set the context for which the search is performed in. I believe that the . at the end of ai. makes the computer treat it as a fully qualified domain name, which causes it to look up the domain name directly, instead of appending the domain search context.
6. dig @a.root-servers.net lirone.csail.mit.edu +norecurs
7. dig . ns
 > a.root-servers.net. 247505 IN A 198.41.0.4
 dig @a.root-servers.net lirone.csail.mit.edu +norecurs
 > a.edu-servers.net. 172800 IN A 192.5.6.30
 dig @a.edu-servers.net lirone.csail.mit.edu +norecurs
 > strawb.mit.edu. 172800 IN A 18.71.0.151
 dig @strawb.mit.edu lirone.csail.mit.edu +norecurs
 > AUTH-NS0.csail.mit.edu. 5336 IN A 128.30.2.123
 dig @AUTH-NS0.csail.mit.edu lirone.csail.mit.edu +norecurs
 > CNAME lirone.lcs.mit.edu.
 dig @AUTH-NS0.csail.mit.edu lirone.lcs.mit.edu +norecurs
 > lirone.lcs.mit.edu. 1800 IN A 18.26.1.36
8. dig www.dmoz.org +norecurs
 The default server did not have the answer; instead it knew the servers for the org domain and it pointed me there. I knew because it did not give me an answer directly.
 >org. 44498 IN NS d0.org.afiliat-nst.org.
 The answer was returned in 0 seconds.
9. dig www.dmoz.org
 The site's IP address was returned in 415 milliseconds.

10. `dig www.dmoz.org +norecurs`

The CNAME record is now returned, but not the site's IP address for some reason, in 0 milliseconds. The cache has served its purpose.

*M.I.T. DEPARTMENT OF EECS***6.033 - Computer System Engineering****WAL Hands-On Assignment**

Hands-on 5: Write Ahead Log System

Intro to wal-sys

Complete the following hands-on assignment. Do the activities described, and submit your solutions using the [online submission site](#) before the **beginning of recitation**.

This hands-on assignment will give you some experience using a Write Ahead Log (WAL) system. This system corresponds to the WAL scheme described in Section 9.3 of the course notes. You should carefully read that section before attempting this assignment. You can do this hands-on on any computer that has a Perl language interpreter, but we will be able to answer your questions more easily if you run this on an Athena workstation. You can download the WAL system from [here](#) (if your browser displays the file in a window instead of saving it, use "File -> Save As" to save the file). The downloaded file is a Perl script named *wal-sys*. Before trying to run it, change its permissions to make it executable, for example by typing:

```
athena% chmod +x wal-sys
```

The *wal-sys* script can be run as follows:

```
athena% wal-sys [-reset]
```

Alternatively, you can run the script as:

```
athena% perl ./wal-sys [-reset]
```

Wal-sys is a simple WAL system that models a bank's central database, implementing redo logging for error-recovery. *Wal-sys* creates and uses two files, named LOG and DB, in the current working directory. The "LOG" file contains the log entries, and the "DB" file contains all of the installed changes to the database.

After you start *wal-sys*, you can enter commands to manage recoverable actions and accounts. There are also commands to simulate a system crash and to print the contents of the "LOG" and "DB" files. All the commands to *wal-sys* are case sensitive. Since *wal-sys* uses the standard input stream, you can use the system in batch mode. To do this, place your commands in a file ("cmd.in" for example) and redirect the file to *wal-sys*'s standard input:

```
athena% wal-sys -reset < cmd.in.
```

When using batch mode, make sure that each command is followed by a newline character (including the last one).

When you restart *wal-sys*, it will perform a log-based recovery of the "DB" file using the "LOG" file it finds in the current working directory. The **-reset** option tells *wal-sys* to discard the contents of any previous "DB" and "LOG" files so that it can start with a clean initial state.

Commands interpreted by wal-sys

The following commands are used for managing recoverable actions and accounts:

- `begin action_id`
Begin a recoverable action denoted by *action_id*. The *action_id* is a positive integer that uniquely identifies a given recoverable action.
- `create_account action_id account_name starting_balance`
Create a new account with the given *account_name* and *starting_balance*. The first argument specifies that this operation is part of recoverable action *action_id*. The *account_name* can be any character string with no white spaces.
- `credit_account action_id account_name credit_amount`
Add *credit_amount* to *account_name*'s balance. This command logs the credit and holds it in a buffer until an `end` command is executed for recoverable action *action_id*.
- `debit_account action_id account_name debit_amount`
Reduce *account_name*'s balance by *debit_amount*. Like `credit`, this command logs the debit and holds it in a buffer until an `end` command is executed for recoverable action *action_id*.
- `commit action_id`
Commit the recoverable action *action_id*. This command logs a commit record.
- `checkpoint`
Log a checkpoint record.
- `end action_id`
End recoverable action *action_id*. This command installs the results of recoverable action *action_id* to the "DB". It also logs an end record.

The following commands help us understand the dynamics of the WAL system:

- `show_state`
Print out the current state of the database. This command displays the contents of the "DB" and "LOG" files.
- `crash`
Crash the system. In this hands-on, we are only concerned about crash recovery, so this is the only command we will use to exit the program.

Using wal-sys

Start *wal-sys* with a reset:

```
athena% wal-sys -reset
```

and run the following commands (sequence 1):

```
begin 1
create_account 1 studentA 1000
```



```

commit 1
end 1
begin 2
create_account 2 studentB 2000
begin 3
create_account 3 studentC 3000
credit_account 3 studentC 100
debit_account 3 studentA 100
commit 3
show_state
crash

```

Wal-sys should print out the contents of the "DB" and "LOG" files, and then exit.

Use a text editor to examine the "DB" and "LOG" files and answer the following questions (do not run *wal-sys* again until you have answered these questions):

Question 1: *Wal-sys* displays the current state of the database contents after you type `show_state`. Why doesn't the database show *studentB*?

Question 2: When the database recovers, which accounts should be active, and what values should they contain?

Question 3: Can you explain why the "DB" file does not contain a record for *studentC* and contains the pre-debit balance for *studentA*?

Recovering the database

When you run *wal-sys* without the `-reset` option it recovers the database "DB" using the "LOG" file. To recover the database and then look at the results, type:

```

athena% wal-sys
> show_state
> crash

```

Question 4: What do you expect the state of "DB" to be after *wal-sys* recovers? Why?

Question 5: Run *wal-sys* again to recover the database. Examine the "DB" file. Does the state of the database match your expectations? Why or why not?

Question 6: During recovery, *wal-sys* reports the *action_ids* of those recoverable actions that are "Losers", "Winners", and "Done". What is the meaning of these categories?

Checkpoints

Start *wal-sys* with a reset:

```
athena% wal-sys -reset
```

and run the following commands (sequence 2):

```

begin 1
create_account 1 studentA 1000
commit 1

```

```
end 1
begin 2
create_account 2 studentB 2000
checkpoint
begin 3
create_account 3 studentC 3000
credit_account 3 studentC 100
debit_account 2 studentB 100
commit 3
show_state
crash
```

Note: the remainder of this assignment is only concerned with sequence 2. We will ask you to crash and recover the system a few times, but you should not run the sequence commands again. (Also note that in sequence 2, the command **debit_account 2 studentB 100** refers to action_id 2, not action_id 3! This is not a typo).

Question 7: Why are the results of recoverable action 2's `create_account 2 studentB 2000` command not installed in "DB" by the `checkpoint` command on the following line?

Examine the "LOG" output file. In particular, inspect the CHECKPOINT entry. Also, count the number of entries in the "LOG" file. Run `wal-sys` again to recover the database.

Question 8: How many lines were rolled back? What is the advantage of using checkpoints?

Note down the *action_ids* of "Winners", "Losers", and "Done". Use the `show_state` command to look at the recovered database and verify that the database recovered correctly. Crash the system, and then run `wal-sys` again to recover the database a second time.

Question 9: Does the second run of the recovery procedure (for sequence 2) restore "DB" to the same state as the first run? What is this property called?

Question 10: Compare the *action_ids* of "Winners", "Losers", and "Done" from the second recovery with those from the first. The lists are different. How does the recovery procedure guarantee the property from Question 9 even though the recovery procedure can change? (Hint: Examine the "LOG" file).

Optional: Wal-sys has a hitherto unmentioned option: if you type `wal-sys -undo` it will perform undo logging and undo recovery. Try the above sequences again with undo logging to see what changes.

Go to [6.033 Home Page](#)

Hands On 5
Logging

3/28

Write Ahead Log System (WAL)

Using Perl

Lahh Perl

like Buzilly

It models a bank server

Can batch in

hmm it did not print DB and LOG

Oh you can view them separately - as files

Oh show state did

What does commit do?

End seems to update

②

Why did it recover differently???

L is a qu on the assignment

Same w/ winner + loser

I was wondering ~~that~~ that ...!

Winners, Losers seem like IDs not just counts

Restart

Checkpoint - what does do?

(? Should I have read more about WALs?)

? No advantage to checkpoints?

Still 3 / 2 / 1

Rerecover — / 2 / 1,3

? atomic

& skipping reading

③ Optimal

Undo does not seem to change anything

↗ we do have undo / old amt,

but only line that seems new is 2 is altered

6.033 Reading
Atomicity Chap

3/28

(Online Chap)

all or nothing - mask failures while interrupting programs

before or after - Coordinating concurrent activities

This chapter is about both

don't want card to charge w/o ship
aka "transaction"

Sweeping simplification - same strategy
for error fixing + coordination

? I missed that chap

	<u>all or nothing</u>	<u>before or after</u>
db	> 1 record	records b/w threads
OS	supervisor call	print queue
bank	withdraw/deposit <u>or</u> no	read then write as 1 action

(2)

(I thought this chap could be about logging...)

Correctness w/ concurrency - ~~data~~ must be same as serial

Atomic - higher layer does not know implementation

How does the system fail?

Undoable

* Never modify the only copy!

- So no cell storage

Instead journal storage

- history of previous versions
- has an outcome record

~~Atomically~~ logs

- logs all in journal
- installs changes in cell storage

Purposes - atomicity (crash recovery)
- archive
- performance (faster to write)
- durability (on 2nd storage medium)

③

Log the update before installing

⓪ (not reading close enough to understand)

Write-ahead logging

A stub-class article from Wikipedia, the free encyclopedia

In computer science, **write-ahead logging (WAL)** is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems.

In a system using WAL, all modifications are written to a log before they are applied. Usually both redo and undo information is stored in the log.

The purpose of this can be illustrated by an example. Imagine a program that is in the middle of performing some operation when the machine it is running on loses power. Upon restart, that program might well need to know whether the operation it was performing succeeded, half-succeeded, or failed. If a write-ahead log were used, the program could check this log and compare what it was supposed to be doing when it unexpectedly lost power to what was actually done. On the basis of this comparison, the program could decide to undo what it had started, complete what it had started, or keep things as they are.

WAL allows updates of a database to be done in-place. Another way to implement atomic updates is with shadow paging, which is not in-place. The main advantage of doing updates in-place is that it reduces the need to modify indexes and block lists.

ARIES is a popular algorithm in the WAL family.

File systems typically use a variant of WAL for at least file system metadata called journaling.

The PostgreSQL database system also uses WAL to provide point-in-time recovery and database replication features^[1].

References

- ¹ ^ "Reliability and the Write-Ahead Log" (<http://www.postgresql.org/docs/9.0/static/wal.html>) .
www.postgresql.org. <http://www.postgresql.org/docs/9.0/static/wal.html>. Retrieved 2011-04-15.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Write-ahead_logging&oldid=479454774"

Categories: Database algorithms | Database stubs

-
- This page was last modified on February 29, 2012 at 07:29.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

6.033 2011 Lecture 17: Logging

Last year

Printed to help hands on

Recall from last time:

Two kinds of atomicity: all-or-nothing, before-or-after

Shadow copy can provide all-or-nothing atomicity

[slide: shadow copy]

Golden rule of atomicity: never modify the only copy!

Typical way to achieve all-or-nothing atomicity.

Works because you can fall back to the old copy in case of failure.

Software can also use all-or-nothing atomicity to abort in case of error.

[slide: shadow copy abort/commit]

Drawbacks of shadow file approach:

- only works for single file (annoying but maybe fixable with shadow dirs)

- copy the entire file for every all-or-nothing action (harder to avoid)

Still, shadow copy is a simple and effective design when it suffices.

Many Unix applications (e.g., text editors) use it, owing to rename.

Today, more general techniques for achieving all-or-nothing atomicity.

[slide: transaction syntax]

Idea: keep a log of all changes, and whether each change commits or aborts.

We will start out with a simple scheme that's all-or-nothing but slow.

Then, we will optimize its performance while preserving atomicity.

Consider our bank account example again.

Two accounts: A and B.

Accounts start out empty.

Run these all-or-nothing actions:

```
begin
```

```
A = 100
```

```
B = 50
```

```
commit
```

```
begin
```

```
A = A - 20
```

```
B = B + 20
```

```
commit
```

```
begin
```

```
A = A + 30
```

```
--CRASH--
```

What goes into the log?

We assign every all-or-nothing action a unique transaction ID.

Need to distinguish multiple actions in progress at the same time.

Two kinds of records in the log:

UPDATE records: both new and old value of some variable.

(we'll see in a bit why we need the old values..)

COMMIT/ABORT records: specify whether that action committed or aborted.

TID	T1	T1	T1	T2	T2	T2	T3
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50	COMMIT	A=80	B=70	COMMIT	A=110

What happens when a program runs now?

begin: allocate a new transaction ID.

write variable: append an entry to the log.

read variable: scan the log looking for last committed value.

[slide: read with a log]

As an aside: how to see your own updates?
 Read uncommitted values from your own tid.
 commit: write a commit record.
 Expectedly, writing a commit record is the "commit point" for action,
 because of the way read works (looks for commit record).
 However, writing log records better be all-or-nothing.
 One approach, from last time: make each record fit within one sector.
 abort: do nothing (could write an abort record, but not strictly needed).
 recover from a crash: do nothing.

Quick demo:
 rm DB LOG
 cat 117-demo.txt
 ./wal-sys < 117-demo.txt
 cat LOG

What's the performance of this log-only approach?

Write performance is probably good: sequential writes, instead of random.
 (Since we aren't using the old values yet, we could have skipped the read.)
 Read performance is terrible: scan the log for every read!
 Crash recovery is instantaneous: nothing to do.

How can we optimize read performance?

Keep both a log and "cell storage".
 Log is just as before: authoritative, provides all-or-nothing atomicity.
 Cell storage: provides fast reads, but cannot provide all-or-nothing.
 [board: log, cell storage; updates going to both, read from cell storage]
 We will say we "log" an update when it's written to the log.
 We will say we "install" an update when it's written to cell storage. *cell*
 [slide: read/write with cell storage]

Two questions we have to answer now:

- how to update both the log and cell storage when an update happens?
- how to recover cell storage from the authoritative log after crash?

Let's look at the above example in our situation.

Log still contains the same things.
 As we're running, maintain cell storage for A and B.
 Except one problem: after crash, A's value in cell storage is wrong.
 Last action aborted (due to crash), but its changes to A are visible.
 We're going to have to repair this in our recovery function.
 Good thing we have the log to provide authoritative information.

Ordering of logging and installing.

Why does this matter?

Because the two together don't have all-or-nothing atomicity.
 Can crash inbetween, so just one of the two might have taken place.
 What happens if we install first and then log?
 If we crash, no idea what happened to cell storage, or how to fix it.
 Bad idea, violates the golden rule ("Never modify the only copy").
 The corresponding rule for logging is the "Write-ahead-log protocol" (WAL).
 ==> Log the update before installing it. <==
 If we crash, log is authoritative and intact, can repair cell storage.
 (You can think of it as not being the only copy, once it's in the log.)

Recovering cell storage.

What happens if we log an update, install it, but then abort/crash?
 Need to undo that installed update.
 Plan: scan log, determine what actions aborted ("losers"), undo them.
 [slide: recover cell storage from log]
 Why do we have to scan backwards?

We must know the outcome of every action in that part of log.

Cell storage must reflect all of those log records (commits, aborts).

Truncating mechanism (assuming no pending actions):

Write a checkpoint record, to save our place in the log.

Flush all cached updates to cell storage.

Truncate log prior to checkpoint record.

(Often log implemented as a series of files, so can delete old log files.)

With pending actions, delete before checkpoint & earliest undecided record.

Back to the log records: why do we need all of those parts?

ID: might need to distinguish between multiple actions at the same time.

Undo: roll back losers, in case we wrote to cell storage before abort/crash.

Redo: apply commits, in case we didn't write to cell storage before commit.

Summary.

Logging is a general technique for achieving all-or-nothing atomicity.

Widely used: databases, file systems, ..

Can achieve reasonable performance with logging.

Writes are always fast: sequential.

Reads can be fast with cell storage.

Key idea 1: write-ahead logging, makes it safe to update cell storage.

Key idea 2: recovery protocol, undo losers / redo winners.

What's coming?

Next Monday: dealing with concurrent actions, before-or-after atomicity.

Cannot deal with external actions as part of an all-or-nothing action.

E.g., dispensing money from an ATM: cannot undo or redo during recovery.

Some hope: we'll talk about distributed transactions next Wednesday.

winners need decision

slides

*begin
commit
abort*

Need to undo newest to oldest.

Also need to find outcome of action before we decide whether to undo.
In our example: done will be {1, 2}, we will set cellStorage[A] to 80.

Quick demo:

```
rm DB LOG
cat 117-demo.txt
./wal-sys -undo < 117-demo.txt
cat LOG
cat DB
./wal-sys -undo
show_state
```

undo stuff never installed

What if the programmer decides to abort explicitly, without crashing?

Use the log to find all update records that were part of this action.

Reset cell storage to old values from those records.

Do we need to write an abort record, or can we skip & pretend we crashed?

What if our example had a software abort instead of a crash?

We might access A again later, and write an update record for it.

After a crash, A will be undone to value before aborted action!

So, need an abort record, to indicate that no undo is necessary.

For the same reason, we need to record abort records after recovery.

[slide: recover with abort logging]

Otherwise, will keep rolling back to point before crashed action.

What if we crash during recovery?

Idempotent: can keep recovering over and over again.

Crash during the undo phase: restarting is OK, will perform same undo.

Crash during logging aborts: restarting is OK, duplicate aborts.

What's the performance going to be like now?

Writes might still be OK (but we do write twice: log & install).

Reads are fast: just look up in cell storage.

Recovery requires scanning the entire log, twice, and performing undo/redo.

Remaining performance problems:

We have to write to disk twice.

Scanning the log will take longer and longer, as the log grows.

Optimization 1: defer installing updates, by storing them in a cache.

[slide: read/write with a cache]

writes can now be fast: just one write, instead of two.

the hope is that variable is modified several times in cache before flush.

reads go through the cache, since cache may contain more up-to-date values.

atomicity problem: cell storage (on disk) may be out-of-date.

is it possible to have changes that should be in cell storage, but aren't?

yes: might not have flushed the latest commits.

is it possible to have changes that shouldn't be in cell storage, but are?

yes: flushed some changes that then aborted (same as before).

during recovery, need to go through and re-apply changes to cell storage.

undo every abort (even if it had an explicit record).

redo every commit.

[slide: recovery for caching]

Don't treat actions with an abort record as "done".

-> there might be leftover changes from them in cell storage.

Re-do any actions that are committed (in the "done" set now).

Optimization 2: truncate the log.

Current design requires log to grow without bound: not practical.

What part of the log can be discarded?

I read the other documents

What is commit vs end of
checkpoint?

Winners is in the book

Recovery procedure

WAL reading: log is authoritative
DB is for fast lookup

Recovery procedure

- ~~restores~~ clears all volatile memory
 - cell storage
 - all in progress actions
- 2 passes

1. Backwards (LIFO)

looks at identity + completion status of
every all or nothing action w/ an outcome record

②

These actions - committed or aborted - winners

2. Forward scan

- does all of the redo actions where outcome = committed
 - So puts all back to cell storage
 - So all committed actions on
-

Sometimes cell storage is non-volatile memory
L plus a cache (but can't rely on it)

So db may contain installs that need to
be ~~rolled~~ undone

So during backwards scan look for things
in progress → losers

L actions that committed and not
- basically last record ≠ END

So backward scan that sees a CHANGE stores
the Undo value

So it's like those actions never started

③

Next forward scan doing reds of what committed

Then logs and END for all losers

So future recoveries ignore it and don't try to redo undo

So winners = ~~completed~~ committed but not ended

losers = things not ended or committed before ;) what I saw

(I still don't really get it...)

Review assignment

What is an END?

END shows that all actions completed

BEGIN

↓
CHANGE

↓

OUTCOME

↓
END

} can combine

} can combine

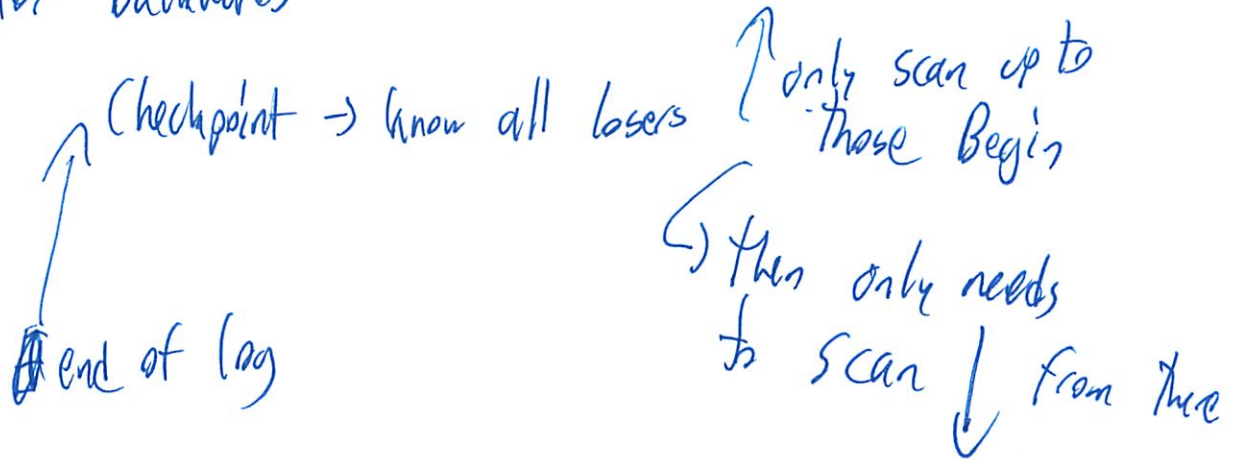
ABORT can roll it back - undo actions
- to BEGIN

(4)

Checkpoint:

- to speed up recovery
- write extra info to non-volatile storage

So for backwards



- greatly speeds up recovery

COMMIT:

- completes the action
- (but is not END)
- they are introduced in separate sections...

So our system only takes action on END

Why does it do this whole recovery thing differently?

Word: idempotent

Michael E Plasmeier

From: Michael E Plasmeier
Sent: Thursday, March 29, 2012 6:24 PM
To: Travis R Grusecki
Subject: Hands on 5
Attachments: Hands On 5 Logging.docx

Travis,

I'm doing hands on 5, but I am pretty confused. I read the book 9.3 and some of the notes from other years, but I am still not 100% sure.

I understand the general operation of a WAL, but I am confused on some of the specifics of this implementation.

I've attached my answers so far. I have some questions:

- Why is there a difference between committed and end?
- Why does the system enforce such a distinction?
 - Why does the WAL recovery process treat processes that are committed but not ended ("winners") any differently from a normal process?
 - Meta question: Why do we have an assignment that focuses on that distinction?
- What purpose does a distinction serve?
- Can you give a real life example of the importance of a distinction?

- Why does the system have this "winner" and "loser" language? Based on my Googling, this language does not seem to be used much beyond 6.033
 - But this language seems connected to the distinction between committed and end.

Thanks! -Michael

Hands-on 5: Logging

Michael Plasmeier

3/29/12
emailed to Travis

1. The database only shows studentA since that was the only transaction that ended with end.
2. With or without the logging system?
Without the logging, only studentA would be there, since that is the only entry in the DB.
With the logging system, all of the actions should be there, including studentA with 900, studentB with 2000, and studentC with 3100.
3. Again, those changes were never ended. This implementation of a WAL only records the values into the DB when end is reached.
4. I would expect that the DB file would stay the same.
5. When it actually runs, we have studentA with 900 and studentC with 3100. This is not what I expected. For some reason, the 3rd transaction has been ended and committed. You can see that in the log.
From reading the textbook, transaction 3 is a "winner" – a committed, but not ended, transaction. The WAL recovery process treats these transactions as ended for some reason.
6. (I was going to ask you that....) Done are transactions committed and ended. Winners transactions that were committed, but never ended. Losers are transactions that are neither committed nor ended. These are undone as the log is read backwards.
7. Checkpoint **does not** write things to DB. It is a step that speeds recovery. (see below)
8. Transactions 1 and 3 were rolled back. Checkpoint writes all of the currently open "losers" at that point in time. When the backwards scan reaches a checkpoint, it only has to scan up to those losers that are listed and no further. This can greatly increase recovery time.
9. The DB is recovered to the same state. This is called *idempotent*.
10. The old one was winners: 3, losers: 2, done: 1. The re-recovery had winners: (nil), losers: 2, done: 1, 3. The log file is updated when the file is recovered to clarify that an action was taken (with END). Losers are ENDED so that future recoveries ignore it and don't try to roll it back..

Michael E Plasmeier

From: Travis R Grusecki
Sent: Thursday, March 29, 2012 9:39 PM
To: Michael E Plasmeier
Subject: RE: Hands on 5

Follow Up Flag: FollowUp
Flag Status: Completed

Michael,

I am a little confused which assignment you are working on, since I don't think we have a hands-on 5 posted online. I think you might be looking at last year's assignment. Either way, I want to answer your questions.

While I can't look at your responses quite yet (I am sitting at an airport), I wanted to give you a quick answer that I think will clarify a lot of your questions. Remember that in a WAL system, entries are made in the log before the associated action occurs, so a COMMIT record doesn't mean that the action has actually been committed to permanent storage. It only means that a decision has been made to commit; the book chooses to call these "winners" – I do think this is mostly only MIT terminology. The END record is written once the commit is complete. At this point, it is safe to remove any record of the transaction from the log because it is entirely finished (on disk). In most situations, a recovery manager will make the following choices depending on the current state:

Recovery:

Before COMMIT: Abort

After COMMIT written: Replay

After END: Do nothing – it has already been stored

I hope this answers some of your questions. I will take a look at your attachment later and try to answer the rest of your questions. Feel free to send more questions. Also, make sure you are working on a current assignment.

-Travis

From: Michael E Plasmeier
Sent: Thursday, March 29, 2012 6:23 PM
To: Travis R Grusecki
Subject: Hands on 5

u/7 My confusion was
that end is never ordered
- it happens

Travis,

I'm doing hands on 5, but I am pretty confused. I read the book 9.3 and some of the notes from other years, but I am still not 100% sure.

I understand the general operation of a WAL, but I am confused on some of the specifics of this implementation.

I've attached my answers so far. I have some questions:

- Why is there a difference between committed and end?
- Why does the system enforce such a distinction?
 - Why does the WAL recovery process treat processes that are committed but not ended ("winners") any differently from a normal process?
 - Meta question: Why do we have an assignment that focuses on that distinction?

Revised 9/7

Hands-on 5: Logging

Michael Plasmeier

1. The database only shows studentA since that was the only transaction that ended with end which means the transaction was written to disk.
2. With or without the logging system?
Without the logging, only studentA would be there, since that is the only entry in the DB.
With the logging system, all of the actions should be there, including studentA with 900, studentB with 2000, and studentC with 3100.
3. Again, those changes were never ended (written to disk). This implementation of a WAL only records the values into the DB when end is reached.
4. I would expect that the DB file would stay the same.
5. When it actually runs, we have studentA with 900 and studentC with 3100. This is not what I expected. For some reason, the 3rd transaction has been ended and committed. You can see that in the log.
From reading the textbook, transaction 3 is a "winner" – it has been ordered committed, but has not yet been written to the disk. The WAL recovery process goes ahead and writes these to disk for us.
6. Done are transactions committed and ended/written to disk. Winners transactions that were ordered committed, but never ended/written to disk. Losers are transactions that are not ordered committed or aborted. These are undone as the log is read backwards.
7. Checkpoint **does not** write things to DB. It is a step that speeds recovery. (see below)
8. Transactions 1 and 3 were rolled back. Checkpoint writes all of the currently open transactions ("losers") at that point in time. When the backwards scan reaches a checkpoint, it only has to scan up to those losers that are listed and no further. This can greatly increase recovery time when most transactions have been ended/written to disk.
9. The DB is recovered to the same state. This is called *idempotent*.
10. The old one was winners: 3, losers: 2, done: 1. The re-recovery had winners: (nil), losers: 2, done: 1, 3. The log file is updated when the file is recovered to clarify that an action was taken (with END). Losers are ENDED so that future recoveries ignore it and don't try to roll it back again in the future.

G.033 2/4

Fault-Tolerance

4/2

Main challenge ^{rest of semester} ~~this unit~~: dealing w/ failure

Had Strong modularity: client/server

- limits propagation
- in 1 computer or internet

But isolates only benign mistakes
and what is the recovery plan?

Can we do better than returning an error?

Can we defend against malicious attacks?

Today: Redundancy/Recovery/Replication

Next 4 lectures: transactions

2 more: Replication state machines

(2)

One set of failures: Window B SOD

Types of failures

- software
- kernel/OS
- machine outage
- design
- operational
 - ↳ BOP codes set wrong
- environmental
 - ↳ earthquake, tsunami

Fault Tolerant Systems

↳ Reliable systems built out of unreliable components

Hard to ensure perfection:

- formal verification?
- is the spec right?
- components may still fail

(3)

Examples

DNS

- if root server fails, 12 more

BGP

- reroutes when a link goes down

TCP

- out of order, duplicate delivery

General Approach:

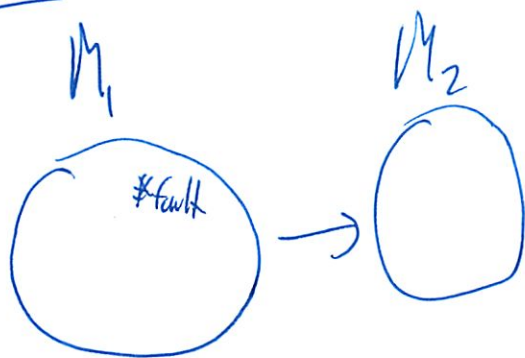
- be careful
- state goals + approaches
- identify all possible faults
 - decide if want to deal w/
 - how to detect
 - how to isolate
- how to handle:
 - fail fast - give an error, don't continue on
 - fail stop
 - might not be able to continue on to return an error

4)

- make - Come up w/ way to handle
like TCP resends
iterative approach

Airplane industry - has black box
- studies crashes
- issues bulletins to look after stuff

Modules



if fault not triggered → latent fault

" " " → active fault

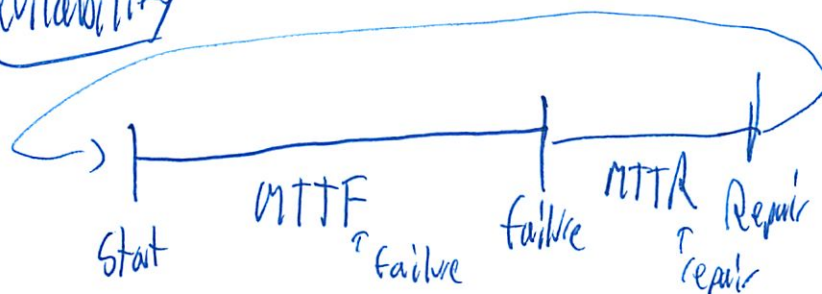
triggers internal error

if M2 can observe → failure

5

Metrics

- availability



$$\text{Availability is } \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

MTTF + MTTR aka ~~MTTB~~ ^{between repair}

(can talk about # of 9s)

99.9993% availability

Reliability of HDD

Commonly fails

persistent storage needed across failures

we have a temp. state in memory

↳ we can recreate it, if needed

(6)

Many systems try to write little to disk
but be able to recreate memory state

How calc MTTF?

It says 13 years - did they actually run that? No!

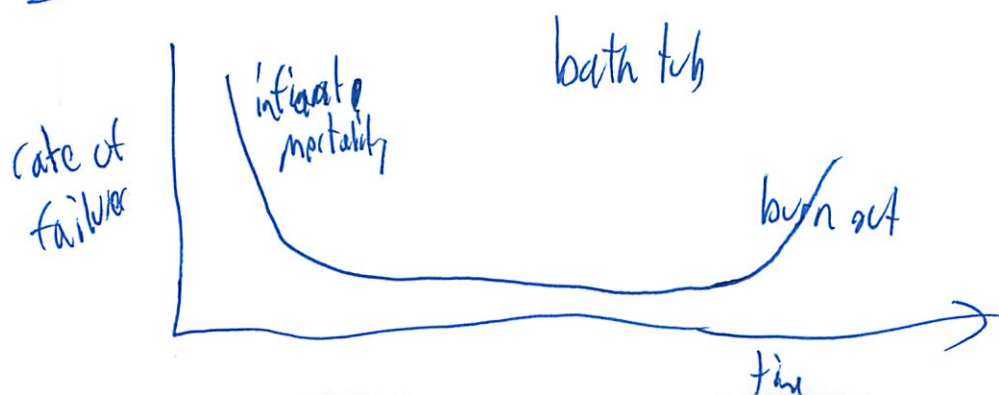
Instead take 1,000 disks
run for 3,000 hrs

If see 10 failures

Then can say $\frac{1}{300,000}$ hrs

$$MTTF = \frac{1}{\text{failure rate}}$$

But failures not \propto likely



(7)

Disk aging is a big issue

Can count # of failures in big data center

We want systems that can deal w/ these kinds of failures

Programs that look at health of disk

Smartctl

- talks to disk controller
- gets stats

disk has a lot of seek errors

- must recalibrate + retry

errors go ↑ at high temp

can see how many blocks "sare" w/ ECC

⑧

Fail fast disk would check checksum

Careful disk would try 10 times

but ECC and data could both be wrong

Could replicate all data

↳ but need double the disks

RADIX does a much better job of this

Lots of ~~complex~~ complexity for truly high reliable

Bit error in SW:

- must write w/ great care

- stringent dev practices

- well-defined specs

- modeling, sim, verification

- N-version programming is tricky

Lots of extra work to really get this right

Rocket system

Like build 2 sep pieces of sw

⑨

But actually the programmers often make mistakes

Same for ~~an~~ secure

↳ getting trusted computing base correct

4/2

```
# -*- mode: org -*-
#+STARTUP: indent
```

6.033 2012 Lecture 14: Fault-tolerant computing

```
* 6.033 so far: client/server
fault isolation by avoiding propagation
  only benign mistakes (programming errors)
  no recovery plan
** rest of semester:
keep running despite failures
broaden class of failures to include malicious ones
```

```
* Threats:
software faults (e.g., blue screen)
hardware (e.g., disk failed)
design (e.g., UI, typo in airport code results in crash, therac-25)
operation (e.g., AT&T outage)
environment (e.g., tsunami)
```

```
* Fault-tolerant systems
Reliable systems form unreliable components
Redundancy (replication, error code, multiple paths)
latence fault -> active fault -> error -> failure -> fault -> etc.
failure = a component fails to produce the intended result at its interface
```

```
* Examples of fault-tolerant systems:
```

```
DNS: replicate NS
Internet: packet network
BGP: alternate path
TCP: resend packet
```

```
* Approach to designing fault tolerant systems:
```

```
1 Identify possible faults
2 Detect & contain (client/server, checksum, etc.)
3 Decide a plan for handling fault:
  nothing
  fail-fast
  fail-safe
  ..
```

```
mask (through redundancy space or time)
Difficult process, because you must identify *all possible* faults
Easy to miss some possibilities, which then bite you later
Iterative process
```

```
* Metric: MTTF and MTBF
MTTF = mean time to failure
MTTR = mean time to repair
MTBF = mean time between failure (MTTF + MTTR)
availability = MTTF / MTBF
```

```
* Examples of availability
See slide
```

```
* Let's look at MTTF of disk and how we can recover
** It allows to store state across power failure
** Building block for fault-tolerant systems:
Store the state that is necessary for recovery on disk
--> All important state stored persistently on disk
Disk better work well
```

```
* MTTF of disk is high!
```

** larger than the existence of the manufacturer
how is it possible?

** run: 1,000 disks for 3,000 hours

10 fail -> 1 failure per 300,000 hours

failure rate = 0.0000033

assume: MTTF = 1/failure rate, then MTTF = 300,000

** is the assumption reasonable? is the failure rate memory less?

* Bathtub graph

Nope; it is not memory less.

Failure rate is compute for bottom part of tub

Is aging a real issue? Yes, see graph.

Has real impact on reliability of system. See graph

* How do we make a fault-tolerant disk

** demo:

ssh root@ud0.csail.mit.edu

smartctl -A /dev/sda > /tmp/smartctl.out

diff /tmp/smartctl.base <(smartctl -A /dev/sda)

.. do something disk-intensive, like du -ks /usr

diff /tmp/smartctl.base <(smartctl -A /dev/sda)

** techniques

1. fail-fast disk (discover faults)

2. retry to handle intermittent failures

3. replication

costs us two disks (see tomorrow)

need to replicate interconnect between disks, controllers, etc. (see HP slide)

* Software complicated -> bugs!

=> Fault-tolerant software systems rely on correctness of software inside disk!

how to write bug-free software

split up in code matter and doesn't mater

most software on computer doesn't matter; code in disk controller does

for code that does matter, stringent development process

hard part: bug in specification

Fault-tolerance

6.033 Lecture 14

Frans Kaashoek

With slides from Sam Madden

Where are we in 6.033?

- Strong form of modularity: client/server
 - Limits propagation of effects
 - In a single computer using OS
 - In a network using Internet
- Two limitations:
 - Isolates only benign mistakes (e.g., programming errors)
 - No recovery plan

Extending C/S to handling failures

- Can we do better than returning an error?
 - Keep computing despite failures?
 - Defend against malicious failures (attacks)?
- Rest of semester: handle these “failures”
 - Fault-tolerant computing
 - Computer security

Plan for fault-tolerant computing

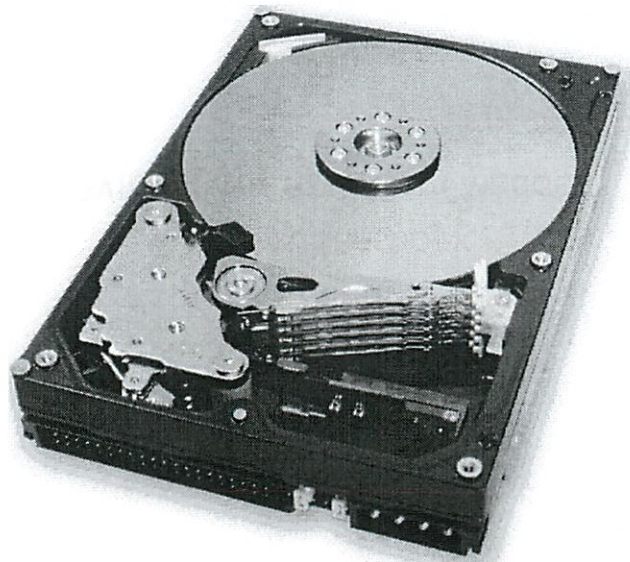
- General introduction: today
 - Redundancy/Recovery/Replication
- Transactions: next 4 lectures
 - updating permanent data in the presence of concurrent actions and failures
- Replication state machines: 2 more
 - Keep computing despite failures

Windows

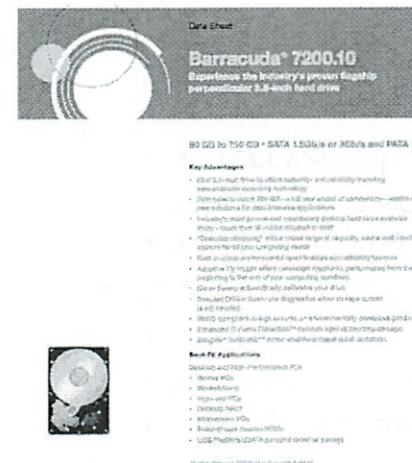
A fatal exception 0E has occurred at 0028:C00068F8 in PPT.EXE<01> + 000059F8. The current application will be terminated.

- * Press any key to terminate the application.
- * Press CTRL+ALT+DEL to restart your computer. You will lose any unsaved information in all applications.

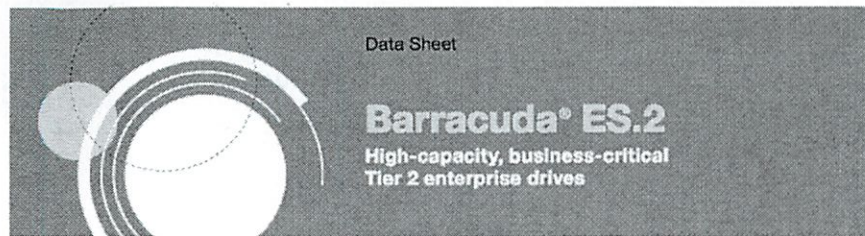
Press any key to continue



- Carrier airlines (2002 FAA fact book)
 - 41 accidents, 6.7M departures
 - ✓ 99.9993% availability
- 911 Phone service (1993 NRIC report)
 - 29 minutes per line per year
 - ✓ 99.994%
- Standard phone service (various sources)
 - 53+ minutes per line per year
 - ✓ 99.99+%
- End-to-end Internet Availability
 - ✓ 95% - 99.6%

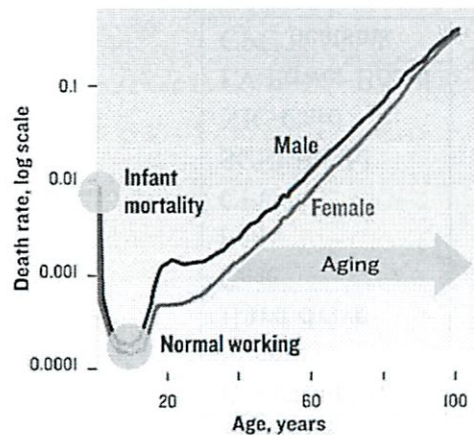
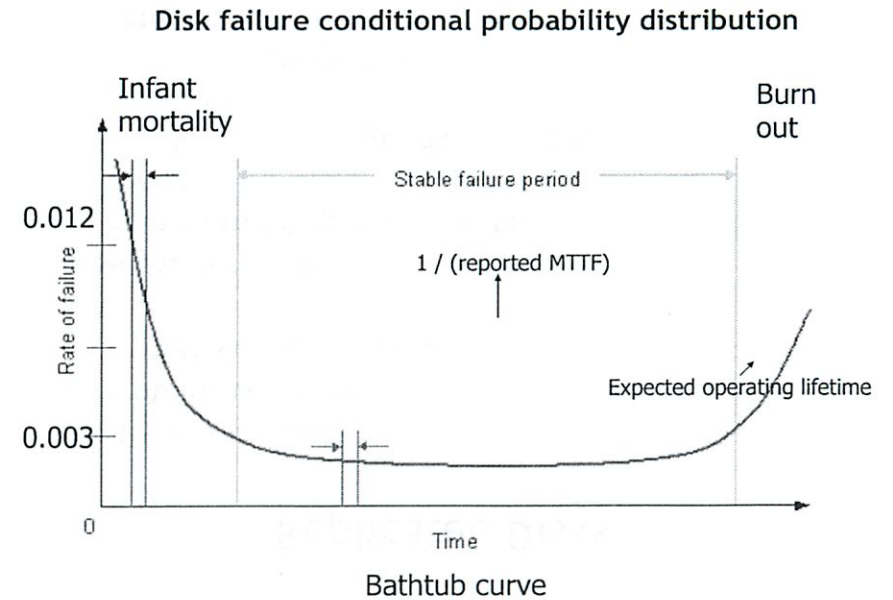


Contact Start-Stops	50,000
Nonrecoverable Read Errors per Bits Read	1 per 10^{14}
Mean Time Between Failures (MTBF, hours)	700,000
Annualized Failure Rate (AFR)	0.34%



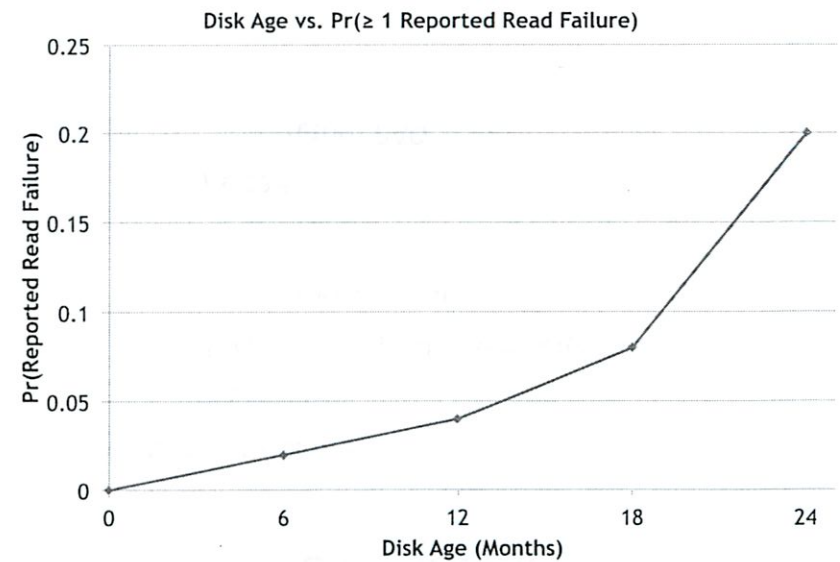
1 TB, 750 GB, 500 GB and 250 GB • 7200 RPM •
SATA 3Gb/s, SATA 1.5Gb/s and SAS 3Gb/s

Reliability/Data Integrity	
Mean Time Between Failures (MTBF, hours)	1.2 million
Reliability Rating at Full 24x7 Operation (AFR)	0.73%
Nonrecoverable Read Errors per Bits Read	1 sector per 10E15
Error Control/Correction (ECC)	10 bit
Interface Ports	
SATA	Single
SAS	Dual



**Human Mortality Rates
(US, 1999)**

From: L. Gavrilov & N. Gavrilova, "Why We Fall Apart," IEEE Spectrum, Sep. 2004.
Data from <http://www.mortality.org>



Bairavasundaram et al., SIGMETRICS 2007

Relative frequency of hardware replacement

COM1	
Component	%
Power supply	34.8
Memory	20.1
Hard drive	18.1
Case	11.4
Fan	8.0
CPU	2.0
SCSI Board	0.6
NIC Card	1.2
LV Power Board	0.6
CPU heatsink	0.6

10,000
machines

Pr(failure in
1 year) $\sim .3$

Schroeder and Gibson, FAST 2008

Fail-fast disk

```
failfast_get (data, sn) {
    get (s, sn);
    if (checksum(s.data) = s.cksum) {
        data ← s.data;
        return OK;
    } else {
        return BAD;
    }
}
```

Careful disk

```
careful_get (data, sn) {
    r ← 0;
    while (r < 10) {
        r ← failfast_get (data, sn);
        if (r = OK) return OK;
        r++;
    }
    return BAD;
}
```

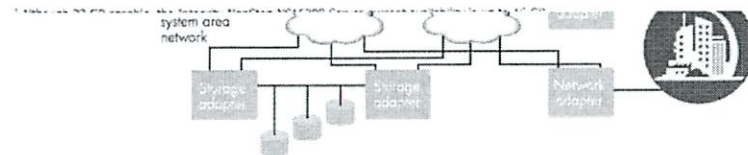
Replicated Disks

```
write (sector, data):
    write(disk1, sector, data)
    write(disk2, sector, data)

read (sector, data):
    data = careful_get(disk1, sector)
    if error
        data = careful_get(disk2, sector)
    if error
        return error
    return data
```


Technical specifications

Processors	2-16 per node Intel Itanium processor 9100 series processors, 1.6 GHz single core processors
Cache	12 MB L3
RAM standard/maximum	Minimum: 4 GB Maximum: 16 GB (32 GB ³)
RAM type/speed	PC2100 ECC registered DDR266A/B
ServerNet I/O	Minimum: 10 Maximum: 60
I/O adapters supported	Fibre Channel, Gigabit Ethernet
Fibre Channel disk modules	14 disks per module
Disk drives supported	146 GB and 300 GB 15K RPM Fibre Channel internal hard disk drives HP Disk Array family (e.g., XP24000, XP20000, XP12000, and XP10000 disk arrays)
Standard features	N + 1 power supplies N + 1 fans



How about an error in software?

- Big problem!
- Software for fault tolerant systems must be written with great care
 - Stringent development practices
 - Well-defined stable specification
 - Modeling, simulation, verification, etc.
 - N-version programming is tricky
- Will also be a problem for secure software
- Good design: small fraction is critical

Read 3/31

Experiences with CoralCDN: A Five-Year Operational View

Michael J. Freedman
Princeton University

Abstract

CoralCDN is a self-organizing web content distribution network (CDN). Publishing through CoralCDN is as simple as making a small change to a URL's hostname; a decentralized DNS layer transparently directs browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin webserver. CoralCDN has been publicly available on PlanetLab since March 2004, accounting for the majority of its bandwidth and serving requests for several million users (client IPs) per day. This paper describes CoralCDN's usage scenarios and a number of experiences drawn from its multi-year deployment. These lessons range from the specific to the general, touching on the Web (APIs, naming, and security), CDNs (robustness and resource management), and virtualized hosting (visibility and control). We identify design aspects and changes that helped CoralCDN succeed, yet also those that proved wrong for its current environment.

1 Introduction

The goal of CoralCDN was to make desired web content available to everybody, regardless of the publisher's own resources or dedicated hosting services. To do so, CoralCDN provides an open, self-organizing web content distribution network (CDN) that any publisher is free to use, without any prior registration, authorization, or special configuration. Publishing through CoralCDN is as simple as appending a suffix to a URL's hostname, e.g., `http://example.com.nyud.net/`. This URL modification may be done by clients, origin servers, or third parties that link to these domains. Clients accessing such Coralized URLs are transparently directed by CoralCDN's network of DNS servers to nearby participating proxies. These proxies, in turn, coordinate to serve content and thus minimize load on origin servers.

CoralCDN was designed to automatically and scalably handle sudden spikes in traffic for new content [14]. It can efficiently discover cached content anywhere in its network, and it dynamically replicates content in proportion to its popularity. Both techniques help minimize origin requests and satisfy changing traffic demands.

While originally designed for decentralized and unmanaged settings, CoralCDN was deployed on the PlanetLab research network [27] in March 2004, given PlanetLab's

convenience and availability. CoralCDN has since remained publicly available for more than five years at hundreds of PlanetLab sites world-wide. Accounting for a majority of public PlanetLab traffic and users, CoralCDN typically serves several terabytes of data per day, in response to tens of millions of HTTP requests from around two million users (unique client IP addresses).

Over the course of its deployment, we have come to acknowledge several realities. On a positive note, CoralCDN's notably simple interface led to widespread and innovative uses. Sites began using CoralCDN as an elastic infrastructure, dynamically redirecting traffic to CoralCDN at times of high resource contention and pulling back as traffic levels abated. On the flip side, fundamental parts of CoralCDN's design were ill-suited for its deployment and the majority of its use. If one were to consider the various reasons for its use—for resurrecting long-unavailable sites, supporting random surfing, distributing popular content, and mitigating flash crowds—CoralCDN's design is insufficient for the first, unnecessary for the second, and overkill for the third, at least given its current deployment. But diverse and unanticipated use is unavoidable for an open system, yet openness is a necessary design choice for handling the final flash-crowd scenario.

This paper provides a retrospective of our experience building and operating CoralCDN over the past five years. Our purpose is threefold. First, after summarizing CoralCDN's published design [14] in Section §2, we present data collected over the system's production deployment and consider its implications. Second, we discuss various deployment challenges we encountered and describe our preferred solutions. Some of these changes we have implemented and incorporated into CoralCDN; others require adoption by third-parties. Third, given these insights, we revisit the problem of building a secure, open, and scalable content distribution network. More specifically, this paper addresses the following topics:

- *The success of CoralCDN's design given observed usage patterns (§3).* Our verdict is mixed: A large majority of its traffic does not require any cooperative caching at all, yet its handling of flash crowds relies on such cooperation.
- *Web security implications of CoralCDN's open API (§4).* Through its open API, sites began leveraging CoralCDN as an elastic resource for content distribution.

? The 1st
CDN?

So this
one is
open?

P2P

? what is PlanetLab

bution. Yet this very openness exposed a number of web security challenges. Many can be attributed to a lack of explicitness for specifying appropriate protection domains, and they arise due to violations of traditional security principles (such as least privilege, complete mediation, and fail-safe defaults [33]).

- *Resource management in CDNs (§5).* CoralCDN commonly faced the challenge of interacting with oversubscribed and ill-behaved resources, both remote origin servers and its own deployment platform. Various aspects of its design react conservatively to change and perform admission control for resources.
- *Desired properties for deployment platforms (§6).* Application deployments could benefit from greater visibility into and control over lower layers of their platforms. Some challenges are again confounded when information and policies cannot be expressed explicitly between layers.
- *Directions for building large-scale, cooperative CDNs (§7).* While using decentralized algorithms, CoralCDN currently operates on a centrally-administered, smaller-scale testbed of trusted servers. We revisit the challenge of escaping this setting.

Rather than focus on CoralCDN’s self-organizing algorithms, the majority of this paper analyzes CoralCDN as an example of an open web service on a virtualized platform. As such, the experiences we detail may have implications to a wider audience, including those developing distributed hash tables (DHTs) for key-value storage, CDNs or web services for elastic provisioning, virtualized network facilities for programmable networks, or cloud computing platforms for virtualized hosting. While many of the observations we report are neither new nor surprising in hindsight, many relate to mistakes, oversights, or limitations of CoralCDN’s original design that only became apparent to us from its deployment.

We next review CoralCDN’s architecture and protocols; a more complete description can be found in [14]. All system details presented after §2 were developed subsequent to that publication. We discuss related work throughout the paper as we touch on different aspects of CoralCDN.

2 Original CoralCDN Design

The Coral Content Distribution Network is composed of three main parts: (1) a network of cooperative HTTP proxies that handle client requests from users, (2) a network of DNS nameservers for nyud.net that map clients to nearby CoralCDN HTTP proxies, and (3) the underlying Coral indexing infrastructure and clustering machinery on which the first two applications are built. This paper consistently refers to the system’s *indexing* layer as Coral, and the entire content distribution system as CoralCDN.

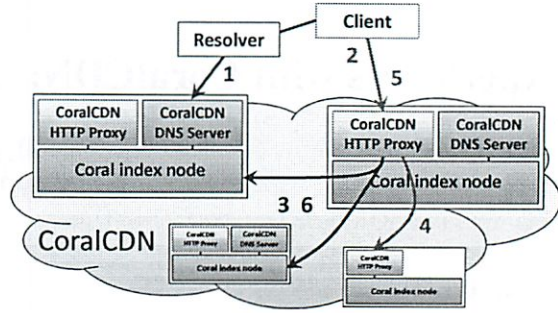


Figure 1: The steps involved in serving a Coralized URL.

2.1 System overview

At a high level, the following steps occur when a client issues a request to CoralCDN, as shown in Figure 1.

1. **Resolving DNS.** A client resolves a “Coralized” domain name (e.g., of the form example.com.nyud.net) using CoralCDN nameservers. A CoralCDN nameserver probes the client to determine its round-trip-time and uses this information to determine appropriate nameservers and proxies to return.
2. **Processing HTTP client requests.** The client sends an HTTP request for a Coralized URL to one of the returned proxies. If the proxy is caching the web object locally, it returns the object and the client is finished. Otherwise, the proxy attempts to find the object on another CoralCDN proxy.
3. **Discovering cooperative-cached content.** The proxy looks up the object’s URL in the Coral indexing layer.
4. **Retrieving content.** If Coral returns the address of a node caching the object, the proxy fetches the object from this node. Otherwise, the proxy downloads the object from the origin server example.com.
5. **Serving content to clients.** The proxy stores the web object to disk and returns it to the client browser.
6. **Announcing cached content.** The proxy stores a reference to itself in Coral, recording the fact that is now caching the URL.

This section reviews the design of the Coral indexing layer and the CDN’s proxies, as proposed in [14].

2.2 Coral indexing layer

The Coral indexing layer is closely related to the structure and organization of distributed hash tables like Chord [34] and Kademlia [23], with the latter serving as the basis for its underlying algorithm. The system maps opaque keys onto nodes by hashing their value onto a flat, semantic-free identifier (ID) space; nodes are assigned identifiers in the same ID space. It allows scalable key lookup (in $O(\log(n))$ overlay hops for n -node systems), reorganizes itself upon network membership changes, and provides robust behavior against failure.

Compared to “traditional” DHTs, Coral introduced a few novel techniques that were well-suited for its particular application [13]. Its key-value indexing layer was designed with weaker consistency requirements in mind, and its lookup structure self-organized into a locality-optimized hierarchy of clusters of peers. After all, a client need not discover all proxies caching a particular file, it only needs to find several such proxies, preferably ones nearby. Like most DHTs, Coral exposes *put* and *get* operations, to announce one’s address as caching a web object, and to discover other proxies caching the object associated with a particular URL, respectively. Inserted addresses are soft-state mappings with a time-to-live (TTL) value.

Coral’s *put* and *get* operations are designed to spread load, both within the DHT and across CoralCDN proxies. To *get* the proxy addresses associated with a key k , a node traverses the ID space with iterative RPCs, and it stops upon finding any remote peer storing values for k . This peer need not be the one closest to k (in terms of DHT identifier space distance). To *put* a key/value pair, Coral routes to nodes successively closer to k and stops when finding either (1) the nodes closest to k or (2) one that is experiencing high request rates for k and already is caching several corresponding values (with longer-lived TTLs). It stores the pair at the node closest to k that it managed to reach. These processes prevent tree saturation in the DHT.

To improve locality, these routing operations are not initially performed across the entire global overlay. Instead, each Coral node belongs to several distinct routing structures called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT). The system is parameterized by a fixed hierarchy of clusters with different expected RTT thresholds. Coral’s deployment uses a three-level hierarchy, with level-0 denoting the global cluster and level-2 the most local one. Coral employs distributed algorithms to form localized, stable clusters, which we briefly return to in §5.3.

Every node belongs to one cluster at each level, as in Figure 2. Coral queries nodes in fast clusters before those in slower clusters. This both reduces lookup latency and increases the chance of returning values stored at nearby nodes, which correspond to addresses of nearby proxies.

2.3 The CoralCDN HTTP proxy

CoralCDN seeks to aggressively minimize load on origin servers. This section summarizes how its proxies use Coral for inter-proxy cooperation and adaptation to flash crowds.

2.3.1 Locality-optimized inter-proxy transfers

Each CoralCDN proxy keeps a local cache from which it can immediately fulfill client requests. When a client requests a non-resident URL, CoralCDN proxies attempt to fetch web content from each other, using the Coral indexing layer for discovery. A proxy only contacts a URL’s

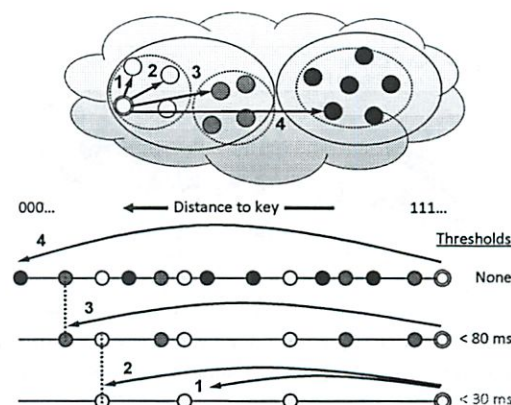


Figure 2: Coral’s three-level hierarchical overlay structure. A node first queries others in its level-2 cluster (the dotted rings), where pointers reference other caching proxies within the same cluster. If a node finds a mapping in its local cluster (after step 2), its *get* finishes. Otherwise, it continues among its level-1 cluster (the solid rings), and finally, if needed, to any node within the global level-0 system.

origin server after the Coral indexing layer provides no referrals or none of its referrals return the data.

CoralCDN’s inter-proxy transfers are optimized for locality, both from their use of parallel connections to other proxies and by the order in which neighboring proxies are contacted. The properties of Coral’s hierarchical indexing ensures that the list of proxies returned by *get* will be sorted based on their cluster distance to the request initiator. Thus, proxies will attempt to contact level-2 neighbors before level-1 and level-0 proxies, respectively.

2.3.2 Rapid adaptation to flash crowds

Unlike many web proxies, CoralCDN is explicitly designed for flash-crowd scenarios. If a flash crowd suddenly arrives for a web object, proxies self-organize into a form of multicast tree for retrieving the object. Data streams from the proxies that started to fetch the object from the origin server to those arriving later. This limits concurrent object requests to the origin server upon a flash crowd.

CoralCDN provides such behavior by cut-through routing and optimistic references. First, CoralCDN’s use of cut-through routing at each proxy helps reduce transmission time for larger files. That is, a proxy will upload portions of a object as soon as they are downloaded, not waiting until it receives the entire object. Second, proxies optimistically announce themselves as sources of content. As soon as a CoralCDN proxy begins receiving the first bytes of a web object—either from the origin or another proxy—it inserts a reference to itself into Coral with a short TTL (30 seconds). It continually renews this short-lived reference until either it completes the download (at which time it inserts a longer-lived reference¹) or the download fails.

¹The deployed system uses 2-hour TTLs for successful results (status codes of 200, 301, 302, etc.), and 15-minute TTLs for 403, 404, and other unsuccessful, non-transient results.

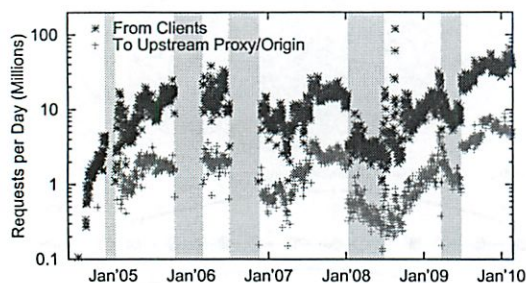


Figure 3: Total HTTP requests per day during CoralCDN’s deployment. Grayed regions correspond to missing or incomplete data.

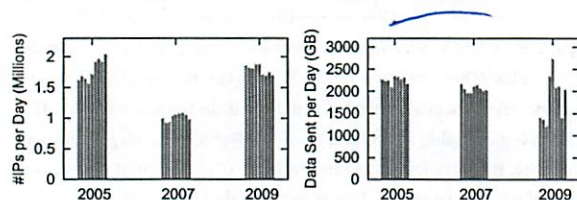


Figure 4: CoralCDN usage: number of unique clients (left) and upload volume (right) for each day during August 9–18.

2.4 Implementation and deployment

CoralCDN is composed of three stand-alone applications. The Coral daemon provides the distributed indexing layer, accessed over UNIX domain sockets from a simple client library linked into applications such as CoralCDN’s HTTP proxy and DNS server. All three are written from scratch. Coral network communication uses Sun RPC over UDP, while CoralCDN proxies transfer content via standard HTTP connections. At initial publication [14], the Coral daemon was about 14,000 lines of C++, the DNS server 2,000 LOC, and the proxy 4,000 LOC. CoralCDN’s implementation has since grown to around 50,000 LOC. The changes we later discuss help account for this increase.

CoralCDN typically runs on 300–400 PlanetLab servers (about 70–100 of which run its DNS server), spread over 100–200 sites worldwide. It avoids Internet2-only and commercial sites, the latter due to policy decisions that restrict their use for open services. CoralCDN uses no special knowledge of these machines’ locations or connectivity (e.g., GPS coordinates, routing information, etc.). Even though CoralCDN runs on a centrally-managed testbed, its mechanisms remain decentralized and self-organizing. The only use of centralization is for managing software and configuration updates and for controlling run status.

3 Analyzing CoralCDN’s Usage

This section presents some HTTP-level data from CoralCDN’s deployment and considers its implications.

3.1 System traces and traffic patterns

To understand some of the HTTP traffic patterns that CoralCDN sees, we analyzed several datasets in increasing

Year	Unique domains	Unique URLs	% URLs with 1 req	Reqs to most popular URL
2005	7881	577K	54%	697K
2007	21555	588K	59%	410K
2009	20680	1787K	77%	1578K

Figure 5: CoralCDN traffic statistics for an arbitrary day (Aug 9).

depth. Figure 3 plots the total number of HTTP requests that the system received each day from mid-2004 through early 2010, showing both the number of HTTP requests from clients, as well as the number of requests issued to upstream CoralCDN peers or origin sites. The traces show common request rates for much of CoralCDN’s deployment between 5 and 20 million HTTP requests per day, with more recent rates of 40–50 million daily requests.²

We examined three time periods from these logs in more depth, each consisting of HTTP traffic over the same nine-day period (August 9–18) in 2005, 2007, and 2009. CoralCDN received 15–25M requests during each day of these periods. Figure 4 plots the total number of unique client IP addresses from which these requests originated (left) and the aggregate amount of bandwidth uploaded (right). The traces showed 1–2 million clients per day, resulting in a few terabytes of content transferred. We will primarily use the 2009 trace, consisting of 209M requests, in later analysis. Figure 5 provides more information about the traffic patterns, focusing on the first day of each trace.

Figure 6 plots the distribution of requests per unique URL. We see that the number of requests per URL follows a Zipf-like distribution, as common among web caching and proxy networks [5]. Certain URLs are very popular—the so-called “head” of the distribution—such as the most popular one in the Aug-9-2009 trace, which received almost 1.6M requests itself. A large number of URLs—the distribution’s “heavy tail”—receive only a single request.

The datasets also show stability in the most popular URLs and domains over time. In all three datasets, the most popular URL retained that ranking across all nine days. In fact, this URL in the 2007 and 2009 traces belonged to the same domain: a site that uses CoralCDN to distribute rule-set updates for the popular Firefox Adblock browser extension. Exploring this further, Figure 7 uses the 2009 trace to plot the request rate per day for the most popular domains (taking the union of each day’s most popular five domains resulted in nine unique domains). We see that six of the nine domains had stable traffic patterns—they were long-term CoralCDN “customers”—while three varied between two and six orders of magnitude per day. The traffic patterns that we see in these two figures have design implications, which we discuss next.

²The peak of 120M requests on August 21, 2008 corresponds to a short-lived experiment of an academic research project using CoralCDN as a key-value store [15].

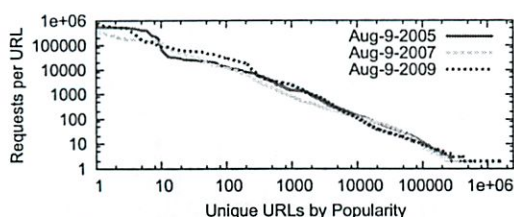


Figure 6: Total requests per unique URL.

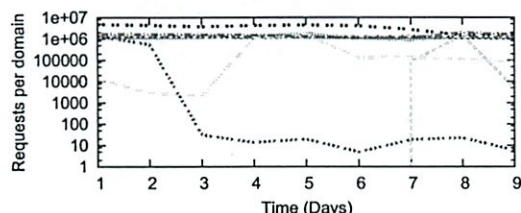


Figure 7: Requests per top-5 domain over time (Aug 9-18, 2009).

3.2 Implications of usage scenarios

For CoralCDN to help under-provisioned websites survive unexpected traffic spikes, it does not require any prior registration or authorization. Yet while such openness is necessary to enable even unmanaged websites to survive flash crowds, it comes at a cost: CoralCDN is used in a variety of ways that differ from this more narrow goal. This section considers how well CoralCDN's design is suited for its four main usage scenarios:

1. **Resurrecting old content:** Anecdotally, some clients attempt to use CoralCDN for long-term durability. One can download browser plugins that link to both CoralCDN and archive.org as potential sources of content when origin servers are unavailable.
2. **Accessing unpopular content:** CoralCDN's request distribution shows a heavy tail of unpopular URLs. Servers may Coralize URLs that few visit. And some clients use CoralCDN as a more traditional proxy, for (presumed) anonymity, censorship or filtering circumvention [32], or automated crawling.
3. **Serving long-term popular content:** Most requests are for a small set of popular objects. These objects, already widely cached across the network, belong to the stable set of customer domains that effectively use CoralCDN as a free, long-term CDN provider.
4. **Surviving flash crowds to content:** Finally, CoralCDN is used for its stated goal of enabling under-provisioned websites to withstand transient load spikes. Popular portals regularly link to Coralized URLs, and users post links in comments. Some sites even adopt dynamic and programmatic mechanisms to redirect requests to CoralCDN, based on observed load and request referrers. We discuss this further in §4.1.

Unfortunately, CoralCDN's design is not well-suited for the first three use cases.

Top URLs	Total Size (MB)	% of Total Reqs
0.01%	14	49.1%
0.1%	157	71.8%
1%	3744	84.8%
10%	28734	92.2%

Figure 8: CoralCDN's working set size for its most popular URLs on Aug 9, 2009: A small percentage of URLs account for a large fraction of requests, yet they require relatively little storage to cache.

Insufficient for resurrecting old content. CoralCDN is not designed for archival storage. Proxies do not proactively replicate content for durability, and unpopular content is evicted from proxy caches over time. Further, if content has an expiry time (default is 12 hours), a proxy will serve expired content for at most 24 hours after the origin fails. Still, some clients attempt to use CoralCDN for this purpose. This underscores a design trade-off: In stressing support for flash crowds rather than long-term durability, CoralCDN devotes its resources to provide availability for content being actively requested. On the other hand, by serving expired content for a limited duration, CoralCDN can mask the *temporary* unavailability of an origin, at least for content already cached in its network.

Unnecessary for unpopular content. While proxies can discover even rare cached content, CoralCDN does not provide any benefit by serving such unpopular content: It does not reduce servers' load meaningfully, and it often results in higher client latency. As such, clients that use CoralCDN to avoid local filtering, circumvent geographic restrictions, or provide (minimal) anonymity may be better served by standard open proxies (that vanilla browsers can be configured to use) or through specialized tools such as Tor [12]. Yet, this type of usage persists—the long tail of Figure 6—and CoralCDN might then be better served with a different design for such traffic, *i.e.*, one that doesn't require a multi-hop, wide-area DHT lookup to complete before fetching content from the origin. For example, for its modest deployment on PlanetLab, each Coral node could maintain connectivity to all others and simply use consistent hashing for a global, one-hop DHT [17, 37]. Alternatively, Coral could only maintain connections with regional peers and eschew global lookups, a design which we evaluate further in §7.

Overkill for stably popular content, so far. For most of CoralCDN's traffic, cooperation is not needed: Figure 6 shows that a small number of URLs accounts for a large fraction of requests. We now measure their working set size in Figure 8, in order to determine how much storage is required to handle this traffic. We find that the most popular 0.01% of URLs account for more than 49% of the total requests to CoralCDN, yet require only 14 MB of storage. Each proxy has a 3.0 GB disk cache, managed using an LRU eviction policy. This is sufficient for serving nearly 85% of all requests from local cache.

web site owner does not have to opt in

connect directly

so instead get the material for it

also what is the design goal for

70.4% hit in local cache
12.6% returned 4xx or 5xx error code
9.9% fetched from origin site
7.1% fetched from other CoralCDN proxy
↳ 1.7% from level-0 cluster (global)
↳ 1.9% from level-1 cluster (regional)
↳ 3.6% from level-2 cluster (local)

Figure 9: CoralCDN access ratios for content during Aug 9, 2009.

These workload distributions support one aspect of CoralCDN's design: Content should be locally cached by the "forward" CoralCDN proxy directly serving end-clients, given that small to moderate size caches in these proxies can serve a very large fraction of requests. This differs from the traditional DHT approach of just storing data on a small number of globally-selected proxies, so-called "server surrogates" [8, 37].

If CoralCDN's working set can be fully cached by each node, we should understand how much cooperation is actually needed. Figure 9 summarizes the extent to which proxies cooperate when handling requests. 70% of requests to proxies are satisfied locally, while only 7% result in cooperative transfers. (The high rate of error messages is due to admission control as a means of bandwidth management, which we discuss in §5.2.) In short, at least for its current workload and environment, only a small fraction of CoralCDN's traffic uses its cooperation mechanisms.

A related result about the limits of cooperative caching had been observed earlier [38], but from the perspective of limited improvements in client-side hit rates. This is a significantly different goal from reducing server-side request rates, however: Non-cooperating groups of nodes would each individually request content from the origin.

This design trade-off comes down to the question of how much traffic is too much for origin servers. For moderately-provisioned origins, such as the customers of commercial CDNs, a caching system might only rely on local or regional cooperation. In fact, Akamai's network is designed precisely so: Nodes within each of its approximately 1000 clusters cooperate, but each cluster typically fetches content independently from origin sites [22]. To replicate such scenarios, Coral's clustering algorithms could be used to self-organize a network into local or regional clusters. It could thus avoid the manual configuration of Harvest [7] or colocated deployments of Akamai.

On the other hand, while cooperation is not needed for most traffic, CoralCDN's ability to react quickly to flash crowds—to offload traffic from a failing or oversubscribed origin—is precisely the scenario for which it was designed (and commercial CDNs are not). We consider these next.

Useful for mitigating flash crowds. CoralCDN's traces regularly show spikes in requests to different URLs. We find, however, that these flash crowds grow in popularity on the order of minutes, not seconds. There is a sufficiently

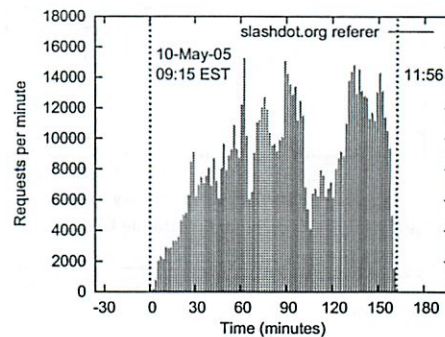


Figure 10: Flash crowd to a Coralized URL linked to by Slashdot.

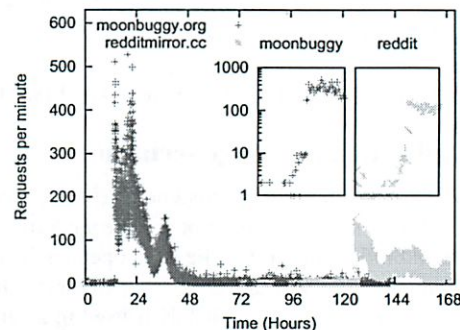


Figure 11: Mini-flash crowds during August 2009 trace. Each datapoint represents a one-minute duration; embedded subfigures show request rates for the tens of minutes around the onset of flash crowds.

long leading edge before traffic rises by several orders of magnitude, which has interesting implications.

Figures 10 and 11 show the request patterns of several flash crowds that CoralCDN experienced. The former was to a site linked to in a Slashdot article in May 2005. After rising, the Slashdot flash crowd lasted less than three hours in duration and came to an abrupt conclusion (perhaps as the story dropped off the website's main page). The latter, covering our August 2009 trace, shows spikes to the image cache of a less popular portal (moonbuggy.org), as well as to a well-publicized mirror for the collaboratively-filtered reddit.com, with another attenuated spike 24 hours later. The embedded graphs in Figure 11 depict the request rates around the onset of the traffic spike for a narrower range of time. All three flash crowds show that the initial rise took minutes.

For a more quantitative analysis of the frequency of flash crowds, we examined the prevalence of domains that experience a large increase in their request rates from one time period to the next. In particular, Figure 12 considers all five-second periods across the August 2009 ten-day trace. The left graph plots a complementary cumulative distribution function (CCDF) of the percentage of domains requested in each period that experience a 10- or 100-fold rate increase. The right graph plots the percentage of requests accounted for by these domains that experience orders-of-magnitude (OOM) increases. Sudden

seems like
not much

when origin
goes down
it's uniquely
good at

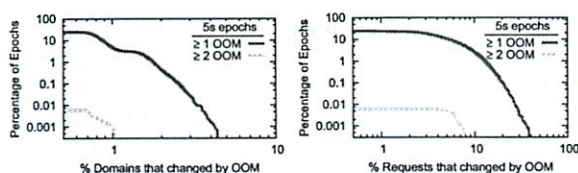


Figure 12: CCDF of extent of flash-crowd dynamics in August 2009 trace. *Left* graph shows percentage of domains experiencing orders of magnitude (OOM) changes in request rates across five-second epochs. *Right* shows % requests for which these domains account.

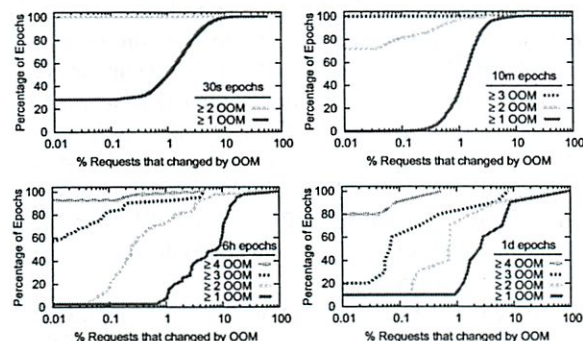


Figure 13: CDFs of percentage of requests accounted for by domains experiencing order(s)-of-magnitude rate increases. Rate increases computed across epochs of 30 seconds (*top left*), 10 minutes (*top right*), six hours (*bottom left*), and one day (*bottom right*). Plots start on the y-axis with zero domains having such an increase, e.g., 28% of 30s epochs have no domains with a ≥ 1 OOM rate increase.

increases do exist, but they are rare. In 76% of 5s epochs, no domains experienced any 10-fold increase, while in 1% of epochs, 1.7% of domains (accounting for 12.9% of requests) increased by one order-of-magnitude. Larger dynamism was even more rare: only in 0.006% of epochs did there exist a domain that experienced a 100-fold increase in request rate. No three OOM increase occurred.

To further understand the precipitousness of “flash” crowds, Figure 13 extends this analysis across longer durations.³ Among 30s epochs, 50% of epochs have at most 0.4% of domains experience a 10-fold increase in their rates (not shown), which account for a total of 1.0% of requests (top left). Only 0.29% of 30s epochs have any domains with more than a 100-fold rate increase. At 10-minute epochs, 28% of epochs have at least one domain that experiences a two OOM rate increase, while 0.21% have a domain with a three OOM increase. Still, these flash crowds account for a small fraction of total requests: Domains experiencing 100-fold increases accounted for at least 1% of all requests in only 3.8% of 10m epochs, and 10% of requests in 0.05% of epochs.

³To avoid overcounting unpopular domains, we do not count changes when the absolute number of requests to a domain in a given time period is less than some minimum amount, i.e., 10 requests for 5s, 30s, and 10m periods, and 100 requests for 6h and 1d periods.

In short, this data shows that (1) only a small fraction of CoralCDN’s domains experience large rate increases within short time periods, (2) those domains’ traffic accounts for a small fraction of the total requests, and (3) any rate increases very rarely occur on the order of seconds.

This moderate adoption rate avoids the need to introduce even more aggressive content discovery algorithms. Simulated workloads in early experiments (Figure 4 of [14]) showed that under high concurrency, CoralCDN might issue several redundant fetches to an origin server due to a race-like condition in its lookup protocol. If multiple nodes concurrently get the same key which does not yet exist in the index, all concurrent lookups can fail and multiple nodes can contact the origin. This race condition is shared by most applications which use a distributed hash table (both peer-to-peer and datacenter services). But because these traces show that the arrival of user requests happens over a much longer time-scale than a DHT lookup, this race condition does not pose a significant problem.

Note that it is possible to mitigate this condition. While designing a network file system for PlanetLab that supported cooperative caching [2]—meant to quickly distribute a file in preparation for a new experiment—we sought to minimize redundant fetches to the file server. We extended Coral’s insert operation to provide return status information, like test-and-set in shared-memory systems. A single *put+get* both returns the first values it encountered in the DHT, as well as inserts its own values at an appropriate location (for a new key, this would be at its closest node). This optimization comes at a subtle cost, however, as it now optimistically inserts a node’s identity even before that proxy begins downloading the file! If the origin fetch fails—a greater possibility in CoralCDN’s environment than with a managed file server—then the use of these index entries degrades performance. Thus, after using this *put+get* protocol in CoralCDN for several months during 2005, we discontinued its use.

CoralCDN’s openness permits users to quickly leverage its resources under load, and its more complex coordination helps mitigate these flash crowds and mask temporary server unavailability. Yet this very openness led to varied usage, the majority of which does not require CoralCDN’s more complex design. As we will see, this openness also introduces other problems.

4 Lessons for the Web

CoralCDN’s naming technique provides an open API for CDN services that can transparently work for almost any website. Over the course of its deployment, clients and servers have used this API to adopt CoralCDN as an elastic resource for content distribution. Through completely automated means, work can be dynamically expanded out to use CoralCDN when websites require additional band-

width resources, and it can be contracted back when flash crowds abate. In doing so, its use presaged the notion of “surge computing” with public cloud platforms. But these naming techniques and CoralCDN’s open design introduce a number of web security problems, many of which are engendered by a *lack of explicitness for specifying protection domains*. We discuss these issues here.

4.1 An API for elastic CDN services

We believe that the central reason for CoralCDN’s adoption has been its simple user interface and open design.

Interface design. While superficially obvious, CoralCDN’s interface design achieves several important goals:

- **Transparency:** Work with *unmodified, unconfigured, and unaware* web clients and web servers.
- **Deep caching:** Retrieve embedded images or links automatically through CoralCDN when appropriate.
- **Server control:** Not interfere with sites’ ability to perform usage logging or otherwise control how their content is served (*e.g.*, via CoralCDN or directly).
- **Ad-friendly:** Not interfere with third-party advertising, analytics, or other tools incorporated into a site.
- **Forward compatible:** Be amenable to future end-to-end security mechanisms for content integrity or other end-host deployed mechanisms.

Consider an alternative and even simpler interface design [11, 25, 29], in which one embeds origin URLs into the HTTP path, *e.g.*, `http://nyud.net/example.com/`. Not only is HTTP parsing simpler, but nameservers would not need to synthesize DNS records on the fly (unlike our DNS servers for `*.nyud.net`). Unfortunately, while this interface can be used to distribute individual objects, it fails on entire webpages. Any relative links would lack the `example.com` prefix that a proxy needs to identify its origin. One alternative might be to try to rewrite pages to add such links, although *active content* such as javascript makes this notoriously difficult. Further, such active rewriting impedes a site’s control over its content, and it can interfere with analytics and advertisements.

CoralCDN’s approach, however, interprets relative links with respect to a page’s Coralized hostname, and thus transparently requests these objects through it as well. But all absolute URLs continue to point to their origin sites, and third-party advertisements and analytics remain largely unaffected. Further, as CoralCDN does not modify content, content also may be amenable to verification through end-to-end content signatures [30, 35].

In short, it was important for adoption that *site owners retain sufficient control over how their content is displayed and accessed*. In fact, our predicted usage scenario of sites publishing Coralized URLs proved to be less popular than that of dynamic redirection (which we did not foresee).

An API for dynamic adoption. CoralCDN was envisioned with manual URL manipulation in mind, whether by publishers editing HTML, users typing Coralized URLs, or third-parties posting links. After deployment, however, users soon began treating CoralCDN’s interface as an API for accessing CDN services.

On the client side, these techniques included simple browser extensions that offer “right-click” options to Coralize links or that provide a link when a page appears unavailable. They ranged to more complex integration into frameworks like Firefox’s Greasemonkey [21]. Greasemonkey allows third-party developers to write site-specific javascript code that, once installed by users, manipulates a site’s HTML content (usually through the DOM interface) whenever the user accesses it. Greasemonkey scripts for CoralCDN include those that automatically rewrite links on popular portals, or modify articles to include tooltips or additional links to Coralized URLs. CoralCDN also has been integrated directly into a number of client-side software packages for podcasting.

The more interesting cases of CoralCDN integration are on the server-side. One common strategy is for the origin to receive the initial request, but respond with a 302 redirect to a Coralized URL. This can work well even for flash crowds, as the overhead of generating redirects is modest compared to that of actually serving the content.

Generating such redirects can be done by installing a server plugin and writing a few lines of configuration code. For example, the complete dynamic redirection rule using Apache’s `mod_rewrite` plugin is as follows.

```
RewriteEngine on
RewriteCond %{HTTP_USER_AGENT} !^CoralWebPrx
RewriteCond %{QUERY_STRING} !(^|&)coral-no-serve$
RewriteRule ^(.*)$ http://%{HTTP_HOST}.nyud.net
                                     %{REQUEST_URI} [R,L]
```

Still, redirection rules must be crafted carefully. In this example, the second line checks whether the client is a CoralCDN proxy and thus should be served directly. Otherwise, a redirection loop potentially could be formed (although proxies prevent this from happening by checking for potential loops and returning errors if one is found).

Amusingly, some early users during CoralCDN’s deployment caused recursion in a different way—and a form of amplification attack—by submitting URLs with a long string of `nyud.net`’s appended to a domain. Before proxies checked for such conditions, this single request caused a proxy to issue a number of requests, stripping the last instance of `nyud.net` off in each iteration.

While the above rewriting rule applies for all requests, other sites incorporate redirection in more inventive ways, such as only redirecting clients arriving from particular high-traffic referrers:

```
RewriteCond %{HTTP_REFERER} slashdot.org [NC,OR]
RewriteCond %{HTTP_REFERER} digg.com [NC,OR]
RewriteCond %{HTTP_REFERER} blogspot.com [NC]
```


And most interestingly, some sites have even combined such tools with server plugins that monitor server load and bandwidth use, so that their servers only start rewriting requests under high load conditions.

Websites therefore used CoralCDN's naming technique to leverage its CDN resources in an elastic fashion. Based on feedback from users, we expanded this "API" to give sites some simple control over how CoralCDN should handle their requests. For example, web servers can include X-Coral-Control response headers, which are saved as cache meta-data, to specify whether CoralCDN proxies should "redirect home" domains that exceed their bandwidth limits (per §5.2) or just return an error as is standard.

4.2 Security and resource protection

A number of security mechanisms curtailed the misuse of CoralCDN. We highlight the design principle for each.

4.2.1 Limiting functionality

CoralCDN proxies have only ever supported GET and HEAD requests. Many of the attacks for which "open" proxies are infamous [24] are simply not feasible. For example, clients cannot use CoralCDN to POST passwords for brute-force cracking. Proxies do not support CONNECT requests, and thus they cannot be used to send spam as SMTP relays or to forge "From" addresses in web mail. Proxies do not support HTTPS and they delete all HTTP cookies sent in headers. These proxies thus provide minimal application functionality needed to achieve their goals, which is cooperatively serving cacheable content.

CoralCDN's design had several unexpected consequences. Perhaps most interestingly, given CoralCDN's multi-layer caching architecture, attempting to crawl or brute-force attack a website via CoralCDN is quite slow. New or randomly-selected URLs first require a DHT lookup to fail, which serves to delay requests against an origin website, in much the same way that ssh "tar pits" delay responses to failed login attempts. In addition, because CoralCDN only handles explicit Coralized URLs, it cannot be used by simply configuring a vanilla browser's proxy settings. Further, CoralCDN cannot be used to anonymously launch attacks, as it eschews anonymity. Proxies use unique User-Agent strings ("CoralWebPrx") and include their identity in Via headers, and they report an instigating client's IP address to the origin server (in an X-Forwarded-For request header). We can only surmise whether the combination of these properties played some role, but CoralCDN has seen little abuse as a platform for proxying server attacks.

4.2.2 Curtailing excessive resource use

CoralCDN's major limiting resource is aggregate bandwidth. The system employs fair-sharing mechanisms to balance bandwidth consumption between origin domains,

which we discuss further in §5.2. In addition to monitoring server-side consumption, proxies keep a sliding window of client-side usage. Not only do we seek to prevent excessive bandwidth consumption by clients, but also an excessive number of (even small) requests. These are caused typically by server misconfigurations that result in HTTP redirection loops (per §4.1) or by "bot" misuse as part of a brute-force attack. While CoralCDN's limited functionality mitigates such attacks, one notable brute-force login attempt took advantage of poor security at a top-5 website, which used cleartext passwords over GET requests.

Given both its storage and bandwidth limitations, CoralCDN enforces a maximum file size of 50 MB. This has generally prevented clients from using CoralCDN for video distribution, a pragmatic goal when deploying proxies on university-hosted PlanetLab servers. We found that sites attempted to circumvent these limits by omitting Content-Length headers (on connections marked as persistent and without chunked encoding). To ensure compliance, proxies now monitor ongoing transfers and halt (and blacklist) any ones that exceed their limits. This skepticism is needed as proxies interact with potentially untrusted servers, and thus must enforce complete mediation [33] to their resources (in this case, bandwidth).

4.2.3 Blacklisting domains and offloading security

We maintain a global blacklist for blocking access to specified origin domain names. Each proxy regularly fetches and reloads the blacklist. This is a practical, but not fundamental, necessity, employed to prevent CoralCDN's deployment sites from restricting its use. Parties that request blacklisting typically cite one of the following reasons.

Suspected phishing. Websites have been concerned that CoralCDN is—or will be confused with—a phishing site. After all, both appear to be "scraping" content and publish a simulacrum under an alternate domain. The difference, of course, is that CoralCDN is serving the site's content unmodified, yet the web lacks any protocol to authenticate the integrity of content (as in S-HTTP [30]) in order to verify this. As SSL only authenticates identity, websites must typically include CDNs in their trusted computing base.

Potential copyright violation. Typically following a DMCA take-down notice, third-parties report that copyrighted material may be found on a Coralized domain and want it blocked. This scenario is mitigated by CoralCDN's explicit naming—which preserves the name of the actual origin in question—and by its caching design. Once content is removed from an origin server, it is evicted automatically from CoralCDN in at most 24 hours. This is a natural implication of its goal of handling flash crowds, rather than providing long-term availability.

Circumventing access-control restrictions. Some domains mediate access to their website via IP-based authen-

tication, whereby requests from particular IP prefixes are granted access. This practice is especially common for online academic journals, in order to provide easy access for university subscribers. But open proxies within whitelisted prefixes would enable external clients to circumvent these access-control restrictions.

By offloading policing to their customers, sites unnecessarily enlarge their security perimeter to include their customer's networks. This scenario is common yet unnecessary. Recall that CoralCDN proxies do not hide their identities, and they include the originating client's IP address in standard request headers. Thus, origin sites can retain IP-based authentication while verifying that a request does not originate from outside allowed prefixes.⁴ Sites are just not making use of this information, and thus fail to properly mediate access to their protected resources.⁵

We did encounter some interesting attacks on our domain-based blacklists, akin to fast-flux networks. An adversary created dynamic DNS records for a random domain that pointed to the IP address of a target domain (an online academic journal). The random domain naturally was not blacklisted by CoralCDN, and the content was successfully downloaded from the origin target. Such a circumvention technique would not have worked if the origin site checked either proxy headers (as above) or even just the Host field of the HTTP request. The Host corresponded to the fast-flux attack domain, not that of the journal. Again, this security hole demonstrates a lack of explicit verification and fail-safe defaults [33].

4.3 Security and naming conflation

We argued that CoralCDN's naming provided a powerful API for accessing CDN services. Unfortunately, its technique has serious implications as the Web's Same Origin Policy (SOP) conflates naming with security.

Browsers use domain names for three purposes. (1) Domains specify *where* to retrieve content after they are resolved to IP addresses, precisely how CoralCDN enacts its layer of indirection. (2) Domains provide a human-readable name for *what administrative entity* a client is interacting with (e.g., the "common name" identified in SSL server certificates). (3) Domains specify *what security policies* to enforce on web objects and their interactions.

The Same Origin Policy specifies how scripts and instructions from an origin domain can access and modify

⁴This does not address the corner case in which the original request comes from an IP address within that prefix, while subsequent ones that access the then-cached content do not. This can be handled typically by marking content as not cacheable, or by having a proxy include headers that explicitly specify its client population (i.e., as "open" or by IP prefix).

⁵One might argue that sites use a pure IP-based filtering approach given its ability to be implemented in layer-3 front-end load balancers. But this is not a simple firewall problem, as sites also permit access for individual users that login with the appropriate credentials. The sites with which we communicated implemented such authorization logic either directly in web servers or in complex, layer-7 front-end appliances.

browser state. This policy applies to manipulating cookies, browser windows, frames, and documents, as well as to accessing other URLs via an XMLHttpRequest. At its simplest level, all of these behaviors are only allowed between resources that belong to an identical origin domain. This provides security against sites accessing each others' private information kept in cookies, for example. It also prevents websites that run advertisements (such as Google's AdSense) from easily performing click fraud to pay themselves advertising dollars by programmatically "clicking" on their site's advertisements.⁶

One caveat to the strict definition of an identical origin [18] is that it provides an exception for domains that share the same domain.tld suffix, in that www.example.com can read and set cookies for example.com. This has bad implications for CoralCDN's naming strategy. When example.com is accessed via CoralCDN, it can manipulate all nyud.net cookies, not just those restricted to example.com.nyud.net.⁷ Concerned with the potential privacy violations from this scenario, CoralCDN deletes all cookies from headers.

Unfortunately, many websites now manage cookies via javascript, so cookie information can still "leak" between Coralized domains on the browser. This happens often without a site's knowledge, as sites commonly use a URL's domain.tld without verifying its name. Thus, if the Coralized example.com writes nyud.net cookies, these will be sent to evil.com.nyud.net if the client visits that webpage. Honest CoralCDN proxies will delete these cookies in transit, but attackers can still circumvent this problem. For example, when a client visits evil.com.nyud.net, javascript from that page can access nyud.net cookies, then issue a XmlHttpRequest back to evil.com.nyud.net with cookie information embedded in the URL. Similar attacks are possible against other uses of the SOP, especially as it relates to the ability to access and manipulate the DOM. Note that these attack vectors exist even while CoralCDN operates on fully-trusted nodes, let alone more peer-to-peer environments!

Rather than conclude that CoralCDN's domain manipulation is fundamentally flawed, we argue that better adherence to security principles is needed. Websites are partially at fault because they default access to domain.tld suffixes too readily, as opposed to stripping the minimal number of domain prefixes: a violation of the principle of least information. An alternative solution that embraces least

⁶This is prevented because advertisements like AdSense load in an iframe that the parent document—the third-party website that stands to gain revenue—cannot access, as the frame belongs to a different domain.

⁷Commercial CDNs like Akamai are typically not susceptible to such attacks, as they generally use a separate top-level domains for each customer, as opposed to CoralCDN's suffix-based approach. Unlike CoralCDN's zero configuration, however, such designs require that origins preestablish an operational relationship with their CDN provider and point their domain to the CDN service (e.g., by aliasing their domain to the CDN through CNAME records in DNS).

privilege (and has much better incremental deployability) would be to *allow sources of content to explicitly constrain default security policies*. As one simple example, when serving content for some `origin.tld`, proxies could use HTTP response headers to specify that the most permissive domain should be `origin.tld.domain.tld`, not their own `domain.tld`. Interestingly, HTML 5, Flash, and various javascript hacks [6] are all exploring methods to *expand* explicit cross-domain communication.⁸ Both proposals avow that the SOP is insufficient and should be adapted to support more flexible control through explicit rules; ours just views its corner cases as too permissive, while the other views its implications as too restrictive.

5 Lessons for CDNs

Unlike most commercial counterparts, CoralCDN is designed to interact with overloaded or poorly-behaving origin servers. Further, while commercial systems will grow their networks based on expected use (and hence revenue), the CoralCDN deployment is comprised of volunteer sites with fixed, limited bandwidth. This section describes how we adapted CoralCDN to satisfy these realities.

5.1 Designing for faulty origins

Given its design goals, CoralCDN needs to react to non-crash failures at origin servers as the rule, not the exception. Thus, one design philosophy that has come to govern CoralCDN's behavior is that proxies should accept content conservatively and serve results liberally.

Consider the following, fairly common, situation. A portal runs a story that links to a third-party website, driving a sudden influx of readers to this previously unpopular site. A user then posts a Coralized link to the third-party site as a "comment" to the portal's story, providing an alternate means to fetch the content.

Several scenarios are possible. (1) The website's origin server becomes unavailable before any proxy downloads its content. (2) CoralCDN already has a copy of the content, but requests arrive to it after the content's expiry time has passed. Unfortunately, subsequent HTTP requests to the origin webserver result in failures or errors. (3) Or, CoralCDN's content is again expired, but subsequent requests to the origin yield only partial transfers. CoralCDN employs different mechanisms to handle these failures.

Cache negative service results (#1). CoralCDN may be hit with a flood of requests for an inaccessible URL, e.g., DNS resolution fails, TCP connections timeout, etc. For these situations, proxies maintain a local negative result cache about repeated failures. Otherwise, both proxies and their local DNS resolvers have experienced re-

source exhaustion, given flash crowds to apparently dead sites. (While negative result caching has also long been part of some DNS implementations [19], it is not universal and does not extend to TCP or application-level failures.) While more a usability issue, CoralCDN still receives requests for some Coralized URLs several years after their origins became unavailable.

Serve stale content if origin faulty (#2). CoralCDN seeks to avoid replacing good content with bad. As its proxies mostly obey content expiry times specified in HTTP headers,⁹ if cached content expires, proxies perform a conditional request (`If-Modified-Since`) to revalidate or update expired content. Overloaded origin servers might fail to respond or might return some temporary error condition (data in §7 shows this to occur in about 0.5% of origin requests). Rather than retransmit this error, CoralCDN proxies return the stale content and continue to retain it for future use (for up to 24 hours after it expires).

Prevent truncations through whole-file overwrites (#3). Rather than not responding or returning an error, what if a revalidation yields a truncated transfer? This is not uncommon during a flash crowd, as a CoralCDN proxy will be competing for a webserver's resources. Rather than have proxies lose stale yet complete versions of objects, proxies implement whole-file overwrites in the spirit of AFS [16]. Namely, if a valid web object is already cached, the new version is written to a temporary file. Only after the new version completes downloading and appears valid (based on `Content-Length`) will a proxy replace the old one.

These approaches are not fail-proof, limited by both semantic ambiguity in status directives and inaccuracies with their use. In terms of ambiguity, does a 403 (Forbidden) response code signify that a publisher seeks to make the content unavailable (permanent), or is it caused by a website surpassing its daily bandwidth limits and having requests rejected (temporary)? Does a 404 (File Not Found) code indicate whether the condition is permanent (due to a DMCA take-down notice) or temporary (from a PHP or database error)? On the other hand, the application of status directives can be flawed. We often found websites to report human-readable errors in HTML body content, but with an HTTP status code of 200 (Success). This scenario leads CoralCDN to replace valid content with less useful information. We hypothesize that bad defaults in scripting languages such as PHP are partially to blame. Instead of being fail-safe, the response code defaults to success.

Even if transient errors were properly identified, for how long should CoralCDN serve expired content? HTTP lacks

⁸This is in reaction to the common practice of inserting third-party objects into a document's namespace via `<script>`—and thus sacrificing security protections—as the SOP does not permit a middle ground.

⁹Proxies in our deployment are configured with a *minimum* expiry time of some duration (five minutes), and thus do not recognize `No-Cache` directives as such. Because CoralCDN does not support cookies, SSL bridging, or POSTs, however, many of the privacy concerns associated with caching such content are alleviated.

the ability to specify explicit policy for handling expired content. Akamai defaults to a fail-safe scenario by not returning stale content [22], while CoralCDN seeks to balance this goal with availability under server failures. As opposed to only using the system-wide default of 24 hours, CoralCDN recently enabled its users to explicitly specify their policy through `max-stale` response headers.¹⁰

These examples all point to another lesson that governs CoralCDN’s proxy design: *Maintain the status quo unless improvements are possible.*

Decoupling service dependencies. A similar theme of only improving the status quo governs CoralCDN’s management system. CoralCDN servers query a centralized management point for a number of tasks: to update their overall run status, to start or stop individual service components (HTTP, DNS, DHT), to reinstall or update to a new software version, or to learn shared secrets that provide admission control to its DHT. Although designed for intermittent connectivity, one of CoralCDN’s significant outages came when the management server began misbehaving and returning unexpected information. In response, we adopted what one might call *fail-safe behavior* that accepts updates conservatively, an application of decoupling techniques from fault-tolerant systems. Management information is stored durably on servers, maintaining their status-quo operation (even across local crashes) until well-formed new instructions are received.

5.2 Managing oversubscribed bandwidth

While commercial CDNs and computing platforms often respond to oversubscription by acquiring more capacity, CoralCDN’s deployment on PlanetLab does not have that luxury. Instead, the service must manage its bandwidth consumption within prescribed limits. This adoption of bandwidth limits was spurred on by administrative demands from its deployment sites. Following the Asian tsunami of December 2004, and with YouTube yet to be created, CoralCDN distributed large quantities of amateur videos of the natural disaster. With no bandwidth restrictions on PlanetLab at the time, CoralCDN’s network traffic to the public Internet quickly spiked. PlanetLab sites threatened to pull their servers off the network if such use could not be curtailed. It was agreed that CoralCDN should restrict its usage to approximately 10 GB per day per server (*i.e.*, per PlanetLab sliver).

Several design options exist for limiting bandwidth consumption. A proxy could simply shut down after exceeding a configured daily capacity (as supported by Tor [12]). Or it could rate-limit its traffic to prevent transient congestion (as done by BitTorrent and Tor). But as CoralCDN

¹⁰HTTP/1.1 supports `max-stale` request headers, although we are not aware of their use by any HTTP clients. Further, as proxies often evict expired content from their caches, it is unclear whether such request directives can be typically satisfied.

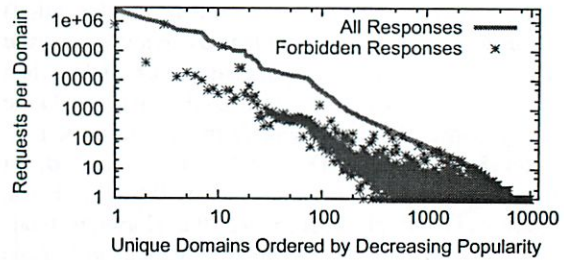


Figure 14: Requests per domain and number of 403 rejections.

primarily provides a service for websites, as opposed to clients, we chose to allocate its limited bandwidth in a way that both preserves some notion of *fairness* across its customer domains and maintains its central goal of handling flash crowds. The technique we developed is more broadly applicable than just PlanetLab and federated testbeds: to P2P deployments where users run peers within resource containers, to multi-tenant datacenters sharing resources between their own services, or to commercial hosting environments using billing models such as 95th-%ile usage.

Providing per-domain fairness might be resource intensive or difficult in the general case, given that CoralCDN interacts with 10,000s of domains each day, but our highly-skewed workloads greatly simplify the necessary accounting. Figure 14 shows the total number of requests per domain that CoralCDN received over one day (the solid top line). The distribution clearly has some very popular domains—the most popular one (a Tamil clone of YouTube) received 2.6M requests—while the remaining distribution fell off in a Zipf-like manner. (Note that Figure 6 was in terms of unique URLs, not unique domains.) Given that CoralCDN’s traffic is dominated by a limited number of domains, its mechanisms can serve mainly to reject requests for (*i.e.*, perform admission control on) these bandwidth hogs. Still, CoralCDN should differentiate between peak limits and steady-state behavior to allow for flash crowds or changing traffic patterns.

To achieve these aims, each CoralCDN proxy implements an algorithm that attempts to simultaneously (1) provide a hard-upper limit on peak traffic per hour (configured to 1000 MB per hour per proxy), (2) bound the expected total traffic per epoch in steady state (400 MB per hour per proxy), and (3) bound the steady-state limit per domain. As setting this last limit statically—such as $1/k$ -th of the total traffic if there are k popular domains—would lead to good fairness but poor utilization (given the non-uniform distribution across domains), we dynamically adjust this last traffic limit to balance this trade-off.

During each hour-long epoch, a proxy records the total number of bytes transmitted for each domain. It also calculates domains’ average bandwidth as an exponentially-weighted moving average (attenuated over one week), as well as the total average consumption across all domains. This long attenuation period provides long-term fairness—

and most consumption is long-term, as shown in Figure 7—but also emphasizes support for short-term flash crowds. Across epochs, bandwidth usage is only tracked, and durably stored, for the top-100 domains. If a domain is not currently one of the top-100 bandwidth consumers, its historical average bandwidth is set to zero (providing additional leeway to sites experiencing flash crowds).

When a requested domain is over its hourly budget (case 3 above), CoralCDN proxies respond with 403 (Forbidden) messages. If instead the proxy is over its peak or steady-state limit calculated over all domains (cases 1 or 2 above), then the proxy redirects the client back to the origin site, and the proxy temporarily makes itself unavailable for new client requests, which would be rejected anyway.¹¹

By applying these mechanisms, CoralCDN reduces its bandwidth consumption to manageable levels. While its demand sometimes exceeds 10 TBs per day (aggregate across all proxies), its actual HTTP traffic remains steady at about 2 TB per day after rejecting a significant number of requests. The scatter plot in Figure 14 shows the number of requests resulting in 403 responses per domain, most due to these admission control mechanisms. We see how variances in domains’ object sizes yield different rejection rates. The second-most popular domain serves mostly images smaller than 10 KB and experiences a rejection rate of 3.3%. Yet the videos of the third-most popular domain—user-contributed screensavers of fractal flames—are typically 5 MB in size, leading to an 89% rejection rate.

Note that we could significantly curtail the use of CoralCDN as a long-term CDN provider (see §3.2) through simple changes to these configuration settings. A low steady-state limit per domain, coupled with a greater weight on a domain’s historic averages, devotes resources to flash-crowd relief at the exclusion of long-term consumption.

Admittedly, CoralCDN’s approach penalizes an origin site with more regional access patterns. Bandwidth accounting and admission control is performed independently on each node, reflecting CoralCDN’s lack of centralization. By not sharing information between nodes (provided that DNS resolution preserves locality), a site with regional interest can be throttled before it reaches its fair share of global capacity. While this does not pose an operational problem for CoralCDN, it is an interesting research problem to perform (approximate) accounting across the network that is both decentralized and scalable. Distributed Rate Limiting [28] considered a related problem, but focused on instantaneous limits (e.g., Mbps) instead of long-term aggregate volumes and gossiped state that is linear in both the number of domains and nodes.

¹¹ If clients are redirected back to the origin, a proxy appends the query-string `coral-no-serve` on the location URL returned to the client. Origins that use redirection scripts with CoralCDN check for this string to prevent loops, per §4.1. Although not the default, operators of some sites preferred this redirection home even if their domain was to blame (a policy they can specify through a `X-Coral-Control` response header).

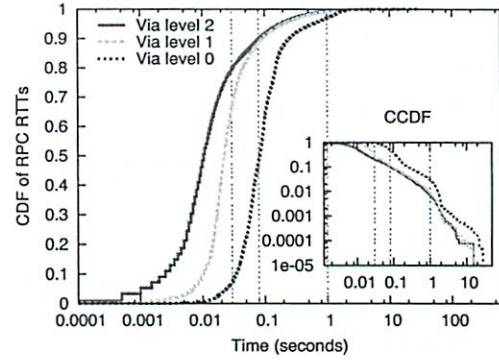


Figure 15: RPC RTTs to various levels of Coral’s DHT hierarchy.

5.3 Managing performance jitter

Running on an oversubscribed deployment platform, CoralCDN developed several techniques to better handle latency variations. With PlanetLab services facing high disk, memory, and CPU contention, and sometimes additional traffic shaping in the kernel, applications can face both performance jitter and prolonged delays. These performance variations are not unique to PlanetLab, and they have been well documented across a variety of settings. For example, Google’s MapReduce [10] took runtime adaption of cluster query processing [3] to the large-scale, where performance variations even among homogeneous components required speculative re-execution of work. More recently, studies of a MapReduce clone on Amazon’s EC2 underscored how shared and virtualized platforms provide new performance challenges [39].

CoralCDN saw the implications of performance variations most strikingly with its latency-sensitive self-organization. For example, Coral’s DHT hierarchy was based on nodes clustering by network RTTs. A node would join a cluster provided some minimum fraction (85%) of its members were below the specified threshold (30 ms for level 2, 80 ms for level 1). Figure 15 shows the RTTs for RPC between Coral nodes, broken down by levels (with vertical lines added at 30ms, 80ms, and 1s). While the clustering algorithms achieve their goals and local clusters have lower RTTs, the heavy tail in all CDFs is rather striking. Fully 1% of RPCs took longer than 1 second, even within local clusters. Coral’s use of concurrent RPCs during DHT operations helped mask this effect.

Another lesson from CoralCDN’s deployment was the need for *stability in the face of performance variations*. This translated to the following rule in Coral. A node would switch to a smaller (and hence less attractive) cluster only if fewer than 70% of a cluster’s members now satisfy its threshold, and form a singleton only if fewer than 50% of neighbors are satisfactory. In other words, the barrier to enter a cluster is high (85%), but once a member, it’s easier to remain. Before leveraging this form of hysteresis, cluster oscillations were much more common, which led

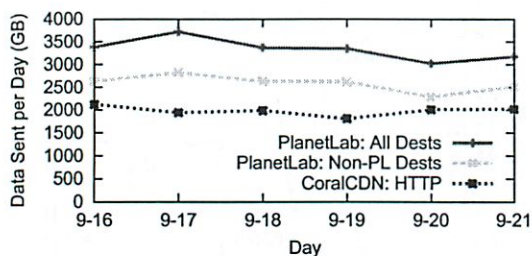


Figure 16: Comparison of PlanetLab's accounting of all upstream traffic, PlanetLab's count to non-PlanetLab destinations, and CoralCDN's accounting through HTTP logs.

to many stale DHT references. A related use of hysteresis within self-organizing systems helped improve virtual network coordinate systems for both PlanetLab [26] and Azureus [20], as well as failure recovery in Bamboo [31].

6 Lessons for Platforms

With the growth of virtualized hosting and cloud deployments, Internet services are increasingly running on third-party infrastructure. Motivated by CoralCDN's deployment on PlanetLab, we discuss some benefits from improving an application's visibility into and control over its lower layers. We first revisit CoralCDN's bandwidth management from the perspective of fine-grained service differentiation, then describe tackling its fault-tolerance challenge with adequate network support.

6.1 Exposing information and expressing preferences across layers

We described CoralCDN's bandwidth management as self-regulating, which works well in trusted environments. But many resource providers would rather *enforce* restrictions than assume applications behave well. Indeed, in 2006, PlanetLab began enforcing average daily bandwidth limits per node per service (*i.e.*, per PlanetLab "sliver"). When a sliver hits 80% of its limit—17.2 GB/day from each server to the public Internet—the kernel begins enforcing bandwidth caps (using Linux's Hierarchical Token Bucket scheduler) as calculated over five-minute epochs.

We now have the possibility of two levels of bandwidth management: admission control by CoralCDN proxies and rate limiting by the underlying hosting platform. Interestingly, even though CoralCDN uses a relatively conservative limit for itself (10 GB/day per sliver), it still surpasses the 80% mark (13.8 GB) on 5–10 servers per day (out of its 300–400 servers). The main cause of this overage is that, while CoralCDN counts only successful HTTP responses, its hosting platform accounts for all traffic—HTTP, DNS, DHT RPCs, log transfers, packet headers, retransmissions, etc.—generated by its sliver. Figure 16 shows the difference in these recorded values for the week of Sept 16, 2009. We see that kernel statistics were 50%–90% higher

than CoralCDN's accounting. This problem of accurate accounting is a general one, as it is difficult or expensive to collect such data in user-space.¹² And even accurate information does not prevent CoralCDN's managed HTTP traffic from competing for network resources with the rest of its sliver's unmanaged traffic.

We argue that hosting platforms should provide better visibility and control. First, these platforms should export greater information to higher levels, such as their current measured resource consumption in a machine-readable format and in real time. Second, these platforms should allow applications to push policies into lower levels, *i.e.*, an application's explicit preferences for handling different classes of resources. For the specific case of network resources, the platform kernel could apply priorities on a granularity finer than just per-sliver, akin to a form of end-host DiffServ; CoralCDN would prioritize DNS and DHT traffic over HTTP traffic, in turn over log maintenance.

Note that we are concerned with a different type of resource management than that provided by VM hypervisors or kernel resource containers [4]. Those systems focus on *short-term* resource isolation or prioritized scheduling between applications, and typically reason about *coarse-grain* VM-level resources. Our focus instead is on *long-term* resource accounting. PlanetLab is not unique here; commercial cloud-computing providers such as Amazon and Rackspace use long-term resource accounting for billing purposes. (In fact, Amazon just launched its CloudWatch service in June 2009 to expose real-time resource monitoring on a coarser-grain per-VM basis [1].) Thus, providing greater visibility and control would be useful not only for deploying applications on platforms with hard constraints (*e.g.*, PlanetLab), but also for managing applications on commercial platforms so as to minimize costs (*e.g.*, in both metered and 95th-%ile billing scenarios).

6.2 Providing support for fault-tolerance

A central reliability issue in CoralCDN is due to its bootstrapping problem: To initially resolve a Coralized URL with no prior knowledge of system participants, a client's resolver must contact one of only 10–12 CoralCDN nameservers registered with the .net gTLD servers. If one of these nameservers fails—each IP address represents a static PlanetLab server—clients experience long DNS timeouts. Thus, while CoralCDN internally detects and reacts quickly to failure, the same rapid recovery is not enjoyed by its primary nameservers registered externally. And once legacy clients bind to a particular proxy's IP address—*e.g.*, web browsers cache name-to-IP mapping to prevent certain types of "rebinding" attacks on the

¹²In fact, even Akamai servers only use an estimate of bandwidth consumption (their so-called "fully-weighted bits") when calculating server load [22]. Only more recently did PlanetLab expose kernel accounting.

Same Origin Policy [9]—CoralCDN cannot recover for this client if that proxy fails.

While certainly observed before, CoralCDN's reliability challenge underscores the limits of purely application-layer recovery, especially as it relates to bootstrapping. In the context of DNS-based bootstrapping, several possibilities exist, including (1) dynamically updating root nameservers to reflect changes, *e.g.*, via the rarely-supported RFC2136 [36], (2) announcing IP anycast addresses via BGP or OSPF, or (3) using transparent network-layer failover between colocated nameservers (*e.g.*, ARP spoofing or VIP/DIP load balancers). IP-level recovery between proxies has its own solutions, but most commonly rely on colocated servers in LAN environments. None of these suggestions are new ones, but they still present a higher barrier to entry; PlanetLab did not have any available to it.

Deployment platforms should strive to provide or expose such network functionality to their services. Amazon EC2's launch of Elastic IP Addresses in March 2008, for example, hid the complexity of ARP spoofing for VM environments. The further development of such support should be an explicit goal for future deployment platforms.

7 Conclusions and Looking Forward

Our retrospective on CoralCDN's deployment has a rather mixed message. We view the adoption of CoralCDN as a successful proof-of-concept of how users can and will leverage open APIs for CDN services. But many of its architectural features were over-designed for its current environment and with its current workload: A much simpler design could have sufficed with probably better performance to boot.

That said, it is a entirely different question as to whether CoralCDN provides a good basis for designing an Internet-scale cooperative CDN. The service remained tied to PlanetLab because we desired a solution that was backwards compatible with both unmodified clients and servers. Running on untrusted nodes seemed imprudent at best given our inability to provide end-to-end security checks. We have shown, however, that even running CoralCDN on fully trusted nodes introduces some security concerns. So, if we dropped the goal of full backwards compatibility, what minimal changes could better support more open, flexible infrastructure?

Naming. CoralCDN's naming provided a layer of indirection for composing two loosely-coupled Internet services. In fact, one could compose longer series of services that each offer different functionality by simply chaining together their domain names. While this technique would not be safe under today's Same Origin Policy, we showed in §4.3 how a trusted proxy could constrain the default security policy. For a participating origin server with an un-

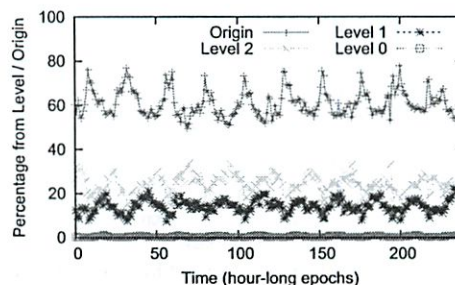


Figure 17: Percentage of a proxy's upstream requests satisfied by origin and by peers at various clustering levels when *regional cooperation* is used, *i.e.*, level-0 peers only serve as a failover from a faulty origin. Dataset covers 10-day period from December 9–19, 2009.

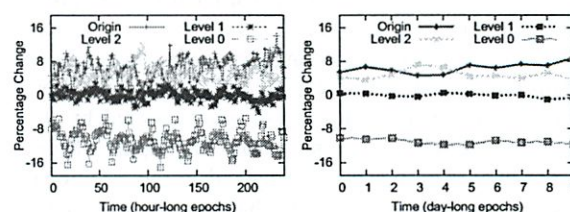


Figure 18: Change in percentage between regional cooperation policy (Figure 17) and CoralCDN's traditional global peering. Positive values correspond to increased hit rates in regional peering.

trusted CDN, the origin should specify (and sign) its minimally required domain suffix of `origin.tld.*`.

Content Integrity. Today's CDNs are full-fledged members of a website's trusted computing base. They have free reign to return modified content. Often, they can even programmatically read and modify any content served *directly* from a customer website to its clients (either by serving embedded `<script>`'s or by playing SOP tricks while masquerading as their customer behind a DNS alias). To provide content delivery via untrusted nodes, the natural solution is an HTTP protocol that supports end-to-end signatures for content integrity [30]. In fact, even a browser extension would suffice to deploy such security [35].

Fine-Grain Origin Control. A tension in this paper is between client latency and server load, underscored by our varied usage scenarios. An appropriate strategy for interacting with a well-provisioned server is a minimal attempt at cooperation before contacting the origin. Yet, an oversubscribed server wants its clients to make a maximal effort at cooperation. So far, proxies have used a "one-size-fits-all" approach, treating all origins as if they were oversubscribed. Instead, much as they have adopted dynamic URL rewriting, origin domains can signal a CoralCDN proxy about their desired policy in-band. At a high-level, this argues for a richer API for elastic CDN services.

To explore the effect of *regional cooperation*, we changed the default lookup policy on about half the deployed CoralCDN proxies since September 2009. If re-

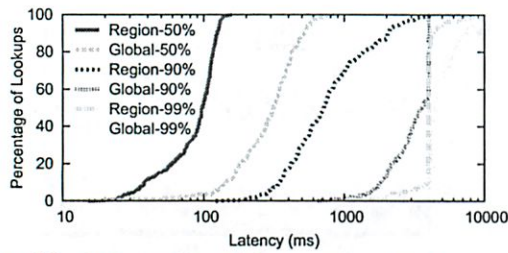


Figure 19: CDF of median, 90th percentile, and 99th percentile lookup latency (over all hour-long epochs of Dec 9–19, 2009), comparing regional and global cooperation policies. Individual lookups were configured with a five-second timeout.

requested content is not already cached locally, these proxies only perform lookups within local and regional clusters (level 2 and 1) before contacting the origin. For proxies operating under such a policy, Figure 17 shows the percentage of upstream requests that were satisfied by the origin and at different levels of clusters. Figure 18 depicts the *change* in behavior compared to the traditional global lookup strategy, showing that the 10–12% of requests that had been satisfied by level-0 proxies shifted to higher hit rates at both the origin and local proxies.¹³ This change was associated with an order-of-magnitude latency improvement for the Coral lookup, shown in Figure 19. The global index still provides some benefit to the system, however, as per Figure 17, it satisfies an average of 0.56% of requests (stddev 0.51%) that failed over from origin servers. In summary, system architectures like CoralCDN can support different policies that trade-off server load for latency, yet still mask temporary failures at origins.

While perhaps imperfectly suited for a smaller-scale platform like PlanetLab, CoralCDN’s architecture provides interesting self-organizational and hierarchical properties. This paper discussed many of the challenges—in security, availability, fault-tolerance, robustness, and, perhaps most significantly, resource management—that we needed to address during its five-year deployment. We believe that its lessons may have wider and more lasting implications for other systems as well.

Acknowledgments. We are grateful to David Mazières for his significant contributions and support during the design and operation of CoralCDN. We also thank Larry Peterson and the entire PlanetLab team for providing a deployment platform for CoralCDN. CoralCDN was originally funded as part of Project IRIS (supported by the NSF under Coop. Agreement #ANI-0225660) and recently under NSF Award #0904860. Freedman was also supported by an NDSEG Fellowship. More information about CoralCDN can be found at www.coralcdn.org.

¹³These graphs also show interesting diurnal patterns, related to a default expiry time of 12 hours for content.

References

- [1] Amazon CloudWatch. <http://aws.amazon.com/cloudwatch/>, 2009.
- [2] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.
- [3] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. Computer Systems*, 21(1), 2003.
- [4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.
- [5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.
- [6] J. Burke. Cross domain frame communication with fragment identifiers. <http://tagneto.blogspot.com/>, June 6, 2006.
- [7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *USENIX Annual*, 1996.
- [8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.
- [9] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from hotjava to netscape and beyond. In *Symp. Security and Privacy*, 1996.
- [10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
- [11] Dijier. <http://code.google.com/p/dijier/>, 2010.
- [12] R. Dingleline, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.
- [13] M. J. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters. In *IPTPS*, 2003.
- [14] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, 2004.
- [15] E. Freudenthal, D. Herrera, S. Gutstein, R. Spring, and L. Longpre. Fern: An updatable authenticated dictionary suitable for distributed caching. In *MMM-ACNS*, 2007.
- [16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1):51–81, 1988.
- [17] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, 1997.
- [18] D. Kristol and L. Montulli. RFC 2965: HTTP state management mechanism, 2000.
- [19] A. Kumar, J. Postel, C. Neuman, P. Danzig, and S. Miller. RFC 1536: Common DNS errors and suggested fixes, 1993.
- [20] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *NSDI*, 2007.
- [21] A. Lieuallen, A. Boodman, and J. Sundstrom. Greasemonkey. <https://addons.mozilla.org/en-US/firefox/addon/748>, 2010.
- [22] B. Maggs. Personal communication, 2009.
- [23] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, 2002.
- [24] V. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy’s view. In *HotNets*, 2003.
- [25] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *NSDI*, 2006.
- [26] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on planetlab. In *WORLDS*, 2005.
- [27] PlanetLab. <http://www.planet-lab.org/>, 2010.
- [28] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.
- [29] RedSwoosh. <http://www.akamai.com/redswoosh>, 2009.
- [30] E. Rescorla and A. Schiffman. RFC 2660: The secure hypertext transfer protocol, 1999.
- [31] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual*, 2004.
- [32] H. Roberts, E. Zuckerman, and J. Palfrey. 2007 circumvention landscape report: Methods, uses, and tools. Technical report, Berkman Center for Internet & Society, Harvard, 2009.
- [33] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 93(9), 1975.
- [34] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Network*, 11(1):17–32, 2003.
- [35] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman. Bringing P2P to the Web: Security and privacy in the Firecoral network. In *IPTPS*, 2009.
- [36] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System, 1997.
- [37] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *OSDI*, Dec 2002.
- [38] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP*, 1999.
- [39] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving map-reduce performance in heterogeneous environments. In *OSDI*, 2008.

DHT

lookup service (hash table) for distributed system
(key, value)

across nodes

so can ~~scale~~ scale
can handle disruptions

Users: BitTorrent, Coral CDN

Wapster had central server

Gnutella sent request to everyone

Freenet put similar keys on certain nodes

Only exact match search

[key space
key space partitioning
overlay network - connects the nodes

②

Can have locality-preserving hashing

↳ Similar keys on similar objects

Then for search → greedy → send to neighbor
whose ID is closest "key-based routing"

↳ want max hops low

Worst case $O(\text{diameter})$

Coral Content Distribution Network

From Wikipedia, the free encyclopedia

The **Coral Content Distribution Network**, sometimes called **Coral Cache** or **Coral**, is a free peer-to-peer content distribution network designed and operated by Michael Freedman. Coral uses the bandwidth of a world-wide network of web proxies and nameservers to mirror web content, often to avoid the Slashdot Effect or to reduce the general load on websites servers in general.

Contents

- 1 Operation
- 2 Usage
- 3 History
- 4 See also
- 5 External links

Coral Content Distribution Network



Developer(s)	Michael Freedman
Initial release	2004
Development status	Active
Operating system	Cross-platform (web-based application)
Type	P2P Web cache
Website	www.coralcdn.org (http://www.coralcdn.org/)

Operation

One of Coral's key goals is to avoid ever creating 'hot spots' of very high traffic, as these might dissuade volunteers from running the software out of a fear that spikes in server load may occur. It achieves this through an indexing abstraction called a distributed sloppy hash table (DSHT); DSHTs create self-organizing clusters of nodes that fetch information from each other to avoid communicating with more distant or heavily-loaded servers.

The *sloppy* hash table refers to the fact that coral is made up of concentric rings of distributed hash tables (DHTs), each ring representing a wider and wider geographic range (or rather, ping range). The DHTs are composed of nodes all within some latency of each other (for example, a ring of nodes within 20 milliseconds of each other). It avoids hot spots (the 'sloppy' part) by only continuing to query progressively larger sized rings if they are not overburdened. In other words, if the two top-most rings are experiencing too much traffic, a node will just ping closer ones: when a node that is overloaded is reached, upward progression stops. This minimises the occurrence of hot spots, with the disadvantage that knowledge of the system as a whole is reduced.

Requests from users are directed to a relatively close node, which then finds the file on the coral DSHT and forwards it to the user.

Usage

A website can be accessed through the Coral Cache by adding `.nyud.net` to the hostname in the site's URL, resulting in what is known as a 'coralized link'. So, for example,

`http://example.com`

becomes

`http://example.com.nyud.net`

Any additional address component after the hostname remains after `.nyud.net`; hence

`http://example.com/folder/page.html`

becomes

`http://example.com.nyud.net/folder/page.html`

For websites that use a non-standard port, for example,

`http://example.com:8080`

becomes

`http://example.com.8080.nyud.net`

History

The project has been deployed since March 2004, during which it has been hosted on PlanetLab, a large-scale distributed research network of several hundred servers deployed at universities world wide. It has not, as originally intended, been deployed by third-party volunteer systems. About 300-400 PlanetLab servers are currently running CoralCDN. The source code is freely available under the terms of the GNU GPL.

Coral Cache gained widespread recognition in the aftermath of the 2004 Indian Ocean earthquake, when it was used to allow access to otherwise inaccessible videos of the resulting tsunami.^{*[citation needed]*}

See also

- CoDeeN
- Globule (CDN)
- Content Delivery Network

External links

- CoralCDN Project (<http://www.coralcdn.org/>)
- Academic paper (NSDI 04) describing CoralCDN (<http://www.coralcdn.org/docs/coral-nsdi04.pdf>)
- Design of CoralCDN (<http://sns.cs.princeton.edu/2009/04/the-design-of-coralcdn/>)
- Michael Freedman's academic homepage (<http://www.cs.princeton.edu/~mfreed/>)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Coral_Content_Distribution_Network&oldid=480543324"

Distributed hash table

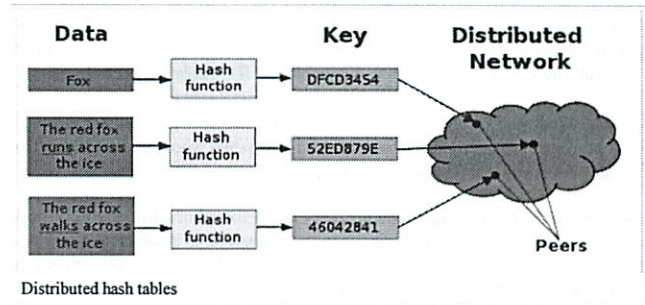
From Wikipedia, the free encyclopedia

A **distributed hash table (DHT)** is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (*key*, *value*) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs form an infrastructure that can be used to build more complex services, such as anycast, cooperative Web caching, distributed file systems, domain name services, instant messaging, multicast, and also peer-to-peer file sharing and content distribution systems. Notable distributed networks that use DHTs include BitTorrent's distributed tracker, the Coral Content Distribution Network, the Kad network, the Storm botnet, and YaCy.

Contents

- 1 History
- 2 Properties
- 3 Structure
 - 3.1 Keyspace partitioning
 - 3.2 Overlay network
 - 3.3 Algorithms for overlay networks
- 4 DHT implementations
- 5 Examples
 - 5.1 DHT protocols and implementations
 - 5.2 Applications employing DHTs
- 6 See also
- 7 References
- 8 External links



History

DHT research was originally motivated, in part, by peer-to-peer systems such as Freenet, gnutella, and Napster, which took advantage of resources distributed across the Internet to provide a single useful application. In particular, they took advantage of increased bandwidth and hard disk capacity to provide a file-sharing service.

These systems differed in how they *found* the data their peers contained:

- Napster, the first large-scale P2P content delivery system to exist, had a central index server: each node, upon joining, would send a list of locally held files to the server, which would perform searches and refer the querier to the nodes that held the results. This central component left the system vulnerable to attacks and lawsuits.
- Gnutella and similar networks moved to a flooding query model—in essence, each search would result in a message being broadcast to every other machine in the network. While avoiding a single point of failure, this method was significantly less efficient than Napster.
- Finally, Freenet is fully distributed, but employs a heuristic key-based routing in which each file is associated with a key, and files with similar keys tend to cluster on a similar set of nodes. Queries are likely to be routed through the network to such a cluster without needing to visit many peers.^[1] However, Freenet does not guarantee that data will be found.

Distributed hash tables use a more structured key-based routing in order to attain both the decentralization of Freenet and gnutella, and the efficiency and guaranteed results of Napster. One drawback is that, like Freenet, DHTs only directly support exact-match search, rather than keyword search, although Freenet's routing algorithm can be generalized to any key type where a closeness operation can be defined.^[2]

In 2001, four systems—CAN, Chord,^[3] Pastry, and Tapestry—ignited DHTs as a popular research topic, and this area of research remains active. Outside academia, DHT technology has been adopted as a component of BitTorrent and in the Coral Content Distribution Network.

Properties

DHTs characteristically emphasize the following properties:

- **Decentralization**: the nodes collectively form the system without any central coordination.
- **Fault tolerance**: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
- **Scalability**: the system should function efficiently even with thousands or millions of nodes.

A key technique used to achieve these goals is that any one node needs to coordinate with only a few other nodes in the system – most commonly, $O(\log n)$ of the n participants (see below) – so that only a limited amount of work needs to be done for each change in membership.

Some DHT designs seek to be secure against malicious participants^[4] and to allow participants to remain anonymous, though this is less common than in many other peer-to-peer (especially file sharing) systems; see anonymous P2P.

Finally, DHTs must deal with more traditional distributed systems issues such as load balancing, data integrity, and performance (in particular, ensuring that operations such as routing and data storage or retrieval complete quickly).

Structure

The structure of a DHT can be decomposed into several main components.^{[5][6]} The foundation is an abstract **keyspace**, such as the set of 160-bit strings. A **keyspace partitioning** scheme splits ownership of this keyspace among the participating nodes. An **overlay network** then connects the nodes, allowing them to find the owner of any given key in the keyspace.

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To store a file with given *filename* and *data* in the DHT, the SHA-1 hash of *filename* is generated, producing a 160-bit key *k*, and a message *put(k, data)* is sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for key *k* as specified by the keyspace partitioning. That node then stores the key and the data. Any other client can then retrieve the contents of the file by again hashing *filename* to produce *k*, and asking any DHT node to find the data associated with *k* with a message *get(k)*. The message will again be routed through the overlay to the node responsible for *k*, which will reply with the stored *data*.

The keyspace partitioning and overlay network components are described below with the goal of capturing the principal ideas common to most DHTs; many designs differ in the details.

Keyspace partitioning

Most DHTs use some variant of consistent hashing to map keys to nodes. This technique employs a function $\delta(k_1, k_2)$ that defines an abstract notion of the *distance* between the keys *k*₁ and *k*₂, which is unrelated to geographical distance or network latency. Each node is assigned a single key called its *identifier* (ID). A node with ID *i*_{*x*} owns all the keys *k*_{*m*} for which *i*_{*x*} is the closest ID, measured according to $\delta(k_m, i_x)$.

Example. The Chord DHT treats keys as points on a circle, and $\delta(k_1, k_2)$ is the distance traveling clockwise around the circle from *k*₁ to *k*₂. Thus, the circular keyspace is split into contiguous segments whose endpoints are the node identifiers. If *i*₁ and *i*₂ are two adjacent IDs, then the node with ID *i*₂ owns all the keys that fall between *i*₁ and *i*₂.

Consistent hashing has the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. Contrast this with a traditional hash table in which addition or removal of one bucket causes nearly the entire keyspace to be remapped. Since any change in ownership typically corresponds to bandwidth-intensive movement of objects stored in the DHT from one node to another, minimizing such reorganization is required to efficiently support high rates of churn (node arrival and failure).

Locality-preserving hashing ensures that similar keys are assigned to similar objects. This can enable a more efficient execution of range queries. Self-Chord^[7] decouples object keys from peer IDs and sorts keys along the ring with a statistical approach based on the swarm intelligence paradigm. Sorting ensures that similar keys are stored by neighbour nodes and that discovery procedures, including range queries, can be performed in logarithmic time.

Overlay network

Each node maintains a set of links to other nodes (its *neighbors* or routing table). Together, these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology.

All DHT topologies share some variant of the most essential property: for any key *k*, each node either has a node ID that owns *k* or has a link to a node whose node ID is *closer* to *k*, in terms of the keyspace distance defined above. It is then easy to route a message to the owner of any key *k* using the following greedy algorithm (that is not necessarily globally optimal): at each step, forward the message to the neighbor whose ID is closest to *k*. When there is no such neighbor, then we must have arrived at the closest node, which is the owner of *k* as defined above. This style of routing is sometimes called key-based routing.

Beyond basic routing correctness, two important constraints on the topology are to guarantee that the maximum number of hops in any route (route length) is low, so that requests complete quickly; and that the maximum number of neighbors of any node (maximum node degree) is low, so that maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree. Some common choices for maximum degree and route length are as follows, where *n* is the number of nodes in the DHT, using Big O notation:

Degree	Route length	Notice
<i>O</i> (1)	<i>O</i> (<i>n</i>)	
<i>O</i> (log <i>n</i>)	<i>O</i> (log <i>n</i> / log(log <i>n</i>))	
<i>O</i> (log <i>n</i>)	<i>O</i> (log <i>n</i>)	most common, but not optimal (degree/route length)
<i>O</i> (1)	<i>O</i> (log <i>n</i>)	
<i>O</i> (√ <i>n</i>)	<i>O</i> (1)	

The most common choice, *O*(log *n*) degree/route length, is not optimal in terms of degree/route length tradeoff, as such topologies typically allow more flexibility in choice of neighbors. Many DHTs use that flexibility to pick neighbors that are close in terms of latency in the physical underlying network.

Maximum route length is closely related to diameter: the maximum number of hops in any shortest path between nodes. Clearly, the network's worst case route length is at least as large as its diameter, so DHTs are limited by the degree/diameter tradeoff^[8] that is fundamental in graph theory. Route length can be greater than diameter, since the greedy routing algorithm may not find shortest paths.^[9]

Algorithms for overlay networks

Aside from routing, there exist many algorithms that exploit the structure of the overlay network for sending a message to all nodes, or a subset of nodes, in a DHT.^[10] These algorithms are used by applications to do overlay multicast, range queries, or to collect statistics. Two systems that are based on this approach are Structella,^[11] which implements flooding and random walks on a Pastry overlay, and DQ-DHT,^[12] which implements a dynamic querying search algorithm over a Chord network.

DHT implementations

Most notable differences encountered in practical instances of DHT implementations include at least the following

- The address space is a parameter of DHT. Several real world DHTs use 128-bit or 160-bit key space
- Some real-world DHTs use hash functions other than SHA-1.
- In the real world the key *k* could be a hash of a file's *content* rather than a hash of a file's *name* to provide content-addressable storage, so that renaming of the file does not prevent users from finding it.
- Some DHTs may also publish objects of different types. For example, key *k* could be the node *ID* and associated data could describe how to contact this node. This allows publication-of-presence information and often used in IM applications, etc. In the simplest case, *ID* is just a random number that is directly used as

key k (so in a 160-bit DHT ID will be a 160-bit number, usually randomly chosen). In some DHTs, publishing of nodes IDs is also used to optimize DHT operations.

- Redundancy can be added to improve reliability. The $(k, data)$ key pair can be stored in more than one node corresponding to the key. Usually, rather than selecting just one node, real world DHT algorithms select i suitable nodes, with i being an implementation-specific parameter of the DHT. In some DHT designs, nodes agree to handle a certain keyspace range, the size of which may be chosen dynamically, rather than hard-coded.
- Some advanced DHTs like Kademlia perform iterative lookups through the DHT first in order to select a set of suitable nodes and send $put(k, data)$ messages only to those nodes, thus drastically reducing useless traffic, since published messages are only sent to nodes that seem suitable for storing the key k ; and iterative lookups cover just a small set of nodes rather than the entire DHT, reducing useless forwarding. In such DHTs, forwarding of $put(k, data)$ messages may only occur as part of a self-healing algorithm: if a target node receives a $put(k, data)$ message, but believes that k is out of its handled range and a closer node (in terms of DHT keyspace) is known, the message is forwarded to that node. Otherwise, data are indexed locally. This leads to a somewhat self-balancing DHT behavior. Of course, such an algorithm requires nodes to publish their presence data in the DHT so the iterative lookups can be performed.

Examples

DHT protocols and implementations

- Apache Cassandra
- BitTorrent DHT (based on Kademlia as provided by Khashmir^[13])
- CAN (Content Addressable Network)
- Chord
- Kademlia
- Pastry
- P-Grid
- Tapestry
- TomP2P

Applications employing DHTs

- BTDig: BitTorrent DHT search engine
- CloudSNAP: a decentralized web application deployment platform
- Codeen: Web caching
- Coral Content Distribution Network
- Dijjer: Freenet-like distribution network
- FAROO: Peer-to-peer Web search engine
- Freenet: A censorship-resistant anonymous network
- GNUnet: Freenet-like distribution network including a DHT implementation
- JXTA: Opensource P2P platform
- maidsafe: C++ implementation of Kademlia, with NAT traversal and crypto libraries. On its home page listed as "Available as a technology licence and a software solution written in cross platform C++."^[14]
- Oracle Coherence: An In Memory Data Grid built on a Java DHT implementation
- Retrosare: a Friend-to-friend network^[15]
- WebSphere eXtreme Scale: proprietary DHT implementation by IBM,^[16] used for object caching
- YaCy: distributed search engine

See also

- membase: a persistent, replicated, clustered distributed object storage system compatible with memcached protocol
- memcached: a high-performance, distributed memory object caching system
- NCache: a high-performance, distributed in-memory object caching system
- prefix hash tree: sophisticated querying over DHTs
- most distributed data stores employ some form of DHT for lookup.

References

- [^] *Searching in a Small World Chapters 1 & 2* (<https://freenetproject.org/papers/lic.pdf>) , <https://freenetproject.org/papers/lic.pdf>, retrieved 2012-01-10
- [^] "Section 5.2.2" (<https://freenetproject.org/papers/ddisr.pdf>) , *A Distributed Decentralized Information Storage and Retrieval System*, <https://freenetproject.org/papers/ddisr.pdf>, retrieved 2012-01-10
- [^] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems (<http://www.cs.berkeley.edu/~istoica/papers/2003/cacm03.pdf>) . In Communications of the ACM, February 2003.
- [^] Guido Urdaneta, Guillaume Pierre and Maarten van Steen. A Survey of DHT Security Techniques (http://www.globule.org/publi/SDST_acmcs2009.html) . ACM Computing Surveys 43(2), January 2011.
- [^] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach (<http://www.wisdom.weizmann.ac.il/~naor/PAPERS/dh.pdf>) . Proc. SPAA, 2003.
- [^] Gurmeet Singh Manku. Dipsea: A Modular Distributed Hash Table (<http://www-db.stanford.edu/~manku/phd/index.html>) . Ph. D. Thesis (Stanford University), August 2004.
- [^] Agostino Forestiero, Emilio Leonardi, Carlo Mastrianni and Michela Meo. Self-Chord: a Bio-Inspired P2P Framework for Self-Organizing Distributed Systems (<http://dx.doi.org/10.1109/TNET.2010.2046745>) . IEEE/ACM Transactions on Networking, 2010.
- [^] *The (Degree,Diameter) Problem for Graphs* (http://maite71.upc.es/grup_de_grafs/table_g.html) , Maite71.upc.es, http://maite71.upc.es/grup_de_grafs/table_g.html, retrieved 2012-01-10
- [^] Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy Neighbor's Neighbor: the Power of Lookahead in Randomized P2P Networks (<http://citeseer.ist.psu.edu/naor04know.html>) . Proc. STOC, 2004.
- [^] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables (<http://www.sics.se/~ali/thesis/>) . KTH-Royal Institute of Technology, 2006.
- [^] Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build Gnutella on a structured overlay? (<http://dx.doi.org/10.1145/972374.972397>) . Computer Communication Review, 2004.
- [^] Domenico Talia and Paolo Trunfio. Enabling Dynamic Querying over Distributed Hash Tables (<http://dx.doi.org/10.1016/j.jpdc.2010.08.012>) . Journal of Parallel and Distributed Computing, 2010.
- [^] Tribler wiki (<http://www.tribler.org/trac/wiki/Khashmir>) retrieved January 2010.
- [^] *maidsafe-dht* (<http://code.google.com/p/maidsafe-dht/>) , Code.google.com, <http://code.google.com/p/maidsafe-dht/>, retrieved 2012-01-10
- [^] Retrosare FAQ (http://retrosare.sourceforge.net/wiki/index.php/Frequently_Asked_Questions#4-1_How_does_RetroShare_know_my_friend.27s_IP, retrieved December 2011
- [^] Billy Newport, IBM Distinguished Engineer (<http://www.devwebsphere.com/devwebsphere/2010/01/implementing-global-indexes-on-websphere-extreme-scale.html>) retrieved October 2010.

External links

- Distributed Hash Tables, Part 1 (<http://linuxjournal.com/article/6797>) by Brandon Wiley.
- Distributed Hash Tables links (<http://deim.urv.cat/~cpairot/dhts.html>) Carles Pairet's Page on DHT and P2P research
- kademlia.scs.cs.nyu.edu (http://web.archive.org/web/*/http://kademlia.scs.cs.nyu.edu/) Archive.org snapshots of kademlia.scs.cs.nyu.edu
- Hazelcast (<http://code.google.com/p/hazelcast/>) open source DHT implementation
- scale4j (<http://code.google.com/p/scale4j/>) highly scalable domain oriented data-distributed platform for java
- IEEE Survey on overlay network schemes (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.4197&rep=rep1&type=pdf>) covering unstructured and structured decentralized overlay networks including DHTs (Chord, Pastry, Tapestry and others) by Eng-Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma and Steve Lim.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Distributed_hash_table&oldid=485134116"

Categories: Distributed data storage | File sharing

-
- This page was last modified on 2 April 2012 at 09:49.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

DP2 out

Can partner w/ anyone w/ Rudolph's citation
pick your partner right

Look in Ted or Tracy!

Local CDN

Content distribution network

Akamai

Store files locally

DHT in Akamai as well

Big start came when Victoria Secret's site went down
when too many people tried to get on

~~Zipf law~~

Zipf law small # popular sites
large # of unpopular sites

Sites get Slash Dotted

Machines on Planet Lab - on university campuses

②

Universities contribute ~5 machines

To study latency, TTL, etc

So universities can match big cos

Prof: thinks this paper is all moaning + groaning
- read up on failures

Akamai charges on peeks
(? what about local video dist?)

Akamai does it by DNS of main site

w/ Coral changes URL

~~http~~ that is served by their special DNS server
it pings you - knows roughly how far
or random
or round robin
before did least loaded

③

Distributed Hash Table

(key, Value) pair

↑
the URL

First checks if the key is local

Kick things out after 24 hrs

If not in local storage, go around to other servers
to see if they had it

Goal is to ↓ trips to ind site

Were debating having them pre-announce that
visiting the site

Paper said many of the uses are not using
the system to its potential

And overkill for flash crowds

Which didn't happen all that often
was only 14MB of popular content

④

Just dump the 14MB on each server!

It was not used as much as expected

Simplist sol is know where the stuff is

Just keep the stuff there

If not working, change the hash

Hash table $O(1)$

DHT $O(\log N)$

Monopoly leveraging by Comcast in XBOX service