Content Distribution Using an Enhanced BitTorrent System

Nirav Dave, Albert Huang, Yuan Shen, Edmund Wong
{ndave, ashuang, yks, elwong}@mit.edu
6.824 Final Project - Spring 2004

Abstract

In the past several years, content distribution has moved from traditional means to the Internet. With the introduction of peer-to-peer (P2P) networks, millions of users now connect and share content, allowing files to be distributed at a much faster rate than ever before. However, bandwidth limitations make larger content much harder to share. If many clients are interested in some large piece of software, many mirror servers are often necessary to handle the load.

With the introduction of BitTorrent and swarm downloading, distribution of large files is now far more efficient. However, despite the decentralized nature of the nodes in a particular swarm, there is still a single point of failure in the system. Additionally, there are several shortcomings that restrict the standard BitTorrent system from becoming a general purpose P2P network.

In this paper, we propose and implement eTorrent, a content-distribution network based around the BitTorrent protocol. We increase the robustness of the system by replicating the single points of failure, the tracking nodes. This is done by use of a dynamic mapping which allows for an elegant failover procedure. In addition, we make the system more capable of handling more standard P2P-like functions, allowing users to more easily share content.

1. Introduction

BitTorrent is a file transfer system originally designed by Bram Cohen in May 2001 to quickly and efficiently distribute content. To achieve this, BitTorrent introduced a new feature to the peer-to-peer market: swarm downloading [2]. In it, clients downloading the file (known as the *swarm*) also upload to other clients, thus shifting the burden of content distribution from the original source onto the interested parties. Clients find nodes from which to download through a centralized node known as the *tracker*, which is used to track clients serving or downloading a particular file.

There are two problems with this approach. Firstly, the tracker becomes a single point of failure for any given file's swarm. Tracker failure or downtime is a major complaint amongst BitTorrent users. When a file's tracker fails, clients can no longer contact the tracker to find new nodes, which eliminates the nodes' ability to actively find new connections. Additionally, it prevents new clients from joining the swarm. Since nodes can exit the system at any time, the swarm's possible efficiency, which comes largely from the number of nodes transferring a file, is capped.

Secondly, the BitTorrent system is not particularly well suited for P2P applications as it is optimized on the distribution of a single file. Traditional P2P applications rely on nodes sharing many different pieces of content. As clients often enter and exit P2P networks, the files available is constantly changing. Because BitTorrent requires a static tracker for every file as well as some means of finding this tracker (in the form of a static `.torrent` file hosted on a website), a general-purpose BitTorrent-based P2P system would require a lot of maintenance in order to handle the dynamic nature of the network. Thus BitTorrent needs a mechanism to dynamically assign trackers to handle the dynamic nature of P2P file sharing.

Other P2P networks are able to decentralize the location of files; however, they do not leverage swarm downloading because they lack a comprehensive list of clients transferring a file. Also, there is neither a built-in incentive for sharing files with other clients nor a system for obtaining or sharing different parts of a file you have or need.

To solve the issues of maintaining a robust BitTorrent P2P system, we introduce eTorrent, a new P2P network that preserves the efficiency of BitTorrent's swarm downloading while increasing the robustness of trackers. Specifically, the two goals of eTorrent are to provide a system that:

1. Provides efficient content distribution in the face of possible node/network failure, whether caused by the dynamic nature of P2P membership or actual hardware failure; and

2. Allows clients to find the node responsible for some content quickly and easily, given a unique content identifier.

We approach our goals by first replicating the task of client tracking across multiple nodes. Replicated trackers periodically synchronize their client information with each other, thus maintaining a consistent view of the swarm. Files are dynamically assigned to trackers using a consistent hashing technique. Along with one-hop routing, which provides to each node a complete list of trackers, consistent hashing provides a simple mechanism for locating trackers and determining file tracking responsibility amongst trackers.

It should be noted that eTorrent does not address the task of searching of content identifiers. It is left to the user to find appropriate the identifiers via some other system (e.g. search engines, dedicated directories).

1.1. Related Work

Although there is an abundance of literature on the web on BitTorrent and P2P network technologies, there has been relatively little work on improving the robustness of trackers.

To help combat tracker failure, a recent proposal was made to extend the BitTorrent protocol to include multiple trackers [3]. However, the proposed design specifications does not indicate how multiple trackers are synchronized, nor does it indicate how replicated trackers are to be managed (whether manually or automatically).

ShareAza [5] is a P2P client that file shares over several existing P2P networks: BitTorrent [2], Gnutella [6] and eDonkey [7]. When a file tracker for BitTorrent fails its tracked copies can still be accessed on the other networks. This adds a level of redundancy to BitTorrent tracked files. However it does not address the key issue; namely, trackers are still single points of failure.

An approach to finding content on BitTorrent is discussed in Metz [8] involving a modified client that searches multiple trackers in order to find the desired files. This methodology addresses robustness by having redundant trackers, but it does not adequately deal with how to dynamically allocate new trackers should the original ones fail. Furthermore, its does not provide sharing of information across trackers.

1.2. Paper Organization

Sections 2 and 3 provide an overview of the design of the eTorrent network. Section 4 discusses the implementation of the nodes and intercommunication methods. The performance of the system is analyzed in section 5. We discuss future improvements that will be made to the system in section 6. Finally, in section 7 we conclude on the state of the eTorrent network.

2. Architecture

In this section, we discuss the general architecture of the eTorrent system: the nodes in the network, the messages passed between nodes, and the mechanism used to assign files to nodes.

2.1. Nodes

Each node on the eTorrent network can serve two different roles: client and tracker. Note that a node can be both a client and a tracker, for either the same file or different files. For clarity we will treat tracker and client as separate nodes.

A *client node* is one that is sharing or downloading files from the network. In particular, a leecher of file f is a client who is currently downloading that file. Similarly, a seed for file f is a client who currently has a complete copy of the file and is only uploading that file.

A *tracker node* maintains a set of *tracks*, one per file for which it is responsible. Each track keeps state of all leeches and seeds for a particular file (e.g. bytes left to download, IP address, port). Information maintained in a track is equivalent to what a single BitTorrent tracker stores. Each tracker node also maintains a global routing table containing all live trackers in the network. This table is discussed in further detail in section 2.3.

2.2. Consistent Hashing

We use a consistent hashing technique to accomplish the task of mapping files to trackers. We assign unique fixed length *identifiers* forming a unified identifier space to both trackers and files. We designate a *tracker identifier* as NodeId(T) for some tracker T. Identifiers for files are referred to as *content identifiers* and are designated ContentId(f) for a file f.

We define the successor of identifier i, succ(i), to be the tracker T with the smallest identifier larger than i. More precisely, $(\forall T' \in \{\text{Trackers}\}.\ i < \mathsf{NodeId}(T) \leq \mathsf{NodeId}(T')) \Rightarrow \mathsf{succ}(i) = T$. If no such T exists (i.e. for $i \geq \mathsf{NodeId}(T')$), $\mathsf{succ}(i) = T$. $\mathsf{NodeId}(T) = \min(\mathsf{NodeId}(T'))$. Similarly, tracker T is the predecessor of identifier i (or pred(i)) if T has the largest node identifier smaller than i.

To determine which trackers are responsible for a given file, we use a sequence of hash functions indexed by the natural numbers to generate *locator identifiers*. We define LocateId(f, s) to be the cryptographic hash of a ContentId(f) for a file f concatenated with a seed number s ($s \in \mathbb{N}$).

A tracker T_n is the n^{th} tracker for a file if $\exists s \in \mathbb{N}$ such that $\mathsf{succ}(\mathsf{LocateId}(f, s)) = \mathsf{NodeId}(T_n)$ and there are exactly $n-1$ other unique trackers in $\{\mathsf{succ}(\mathsf{LocateId}(f, s')) \mid s > s' \in \mathbb{N}\}$. In the case where n is larger than the number of trackers in the system we remove the uniqueness requirement for the trackers.

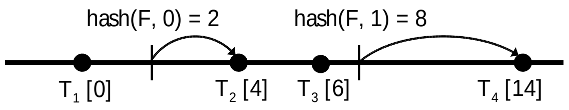

Figure 1. Example of the consistent hashing scheme. T_i are trackers with their identifiers in brackets. F designates a file with two locator identifiers.

We designate that k trackers must be actively tracking any file F in the system. These k trackers are referred to as *co-trackers* in a replicated tracker set. These trackers are in general the first k trackers responsible for a file. However, trackers may refuse to become a tracker for a file due to load issues. In this case, the next assigned tracker will take its place.

Figure 1 shows an example of the tracker-file mapping: file F is assigned to T_2 and T_4 since $\mathsf{succ}(\mathsf{LocateId}(F, 0)) = T_2$ and $\mathsf{succ}(\mathsf{LocateId}(F, 1)) = T_4$.

2.3. One-Hop Routing

Recent publications have suggested a scalable approach to maintaining information amongst nodes in a large network [10]. We use a *one-hop routing* approach [11] to synchronize global information amongst other nodes. In one-hop routing, the nodes form a virtual ring ordered by their node identifiers. This ring is divided into *slices*, which are further divided into *units*. A *slice leader* is elected in each slice by the corresponding nodes.

Each node is responsible for monitoring the status of its two neighbors. Whenever a node detects a new or failed neighbor, a node notifies its slice leader. Each slice leader is responsible for aggregating updates within its slice and exchanging them with other slice leaders. The slice leader then sends all the updates it receives to the median of each unit (called the unit leader [11]), which proceeds to forward it to its neighboring nodes. These updates travel from one neighbor to another until the end of a unit is reached.

The hierarchical passing of events allows routing information to be propagated efficiently to all the nodes. By building our tracker network using one-hop routing, every node has a complete list of all the other trackers, which we call the *global routing table*. Using this list and the consistent hashing algorithm, any tracker in the system can determine the k co-trackers for any file given its content identifier. Thus, a client can connect to any tracker in the system in order to get a list of trackers for the file it is interested in.

Since we use a consistent hashing scheme, we introduce some possible inefficiencies in data passing as trackers exit and enter the system. We choose to do this because the benefits of not having to keep a coherent file-to-tracker table over the entire system greatly overshadow the costs associated with track changes.

2.4. eTorrent Messages

Nodes must be able to communicate membership changes with other nodes in the system in order to maintain the one-hop routing network. Also, trackers in the same replicated tracker set must be able to communicate with co-trackers regarding new clients, etc. Thus, we define a series of messages that can be used in order for nodes to communicate. These messages are summarized in Table 1.

3. Tasks

In this section we describe how the client and tracker nodes interact with other nodes in the eTorrent network in order to accomplish some common operations.

3.1. Contacting the Tracker Network

For a client C to join the eTorrent network, it must contact the tracker network. Client C must already have a list of previously known active trackers on the tracker network. C attempts to contact trackers one-by-one on it's list until it successfully contacts a live tracker G_C or exhausts the list. G_C is referred to as C's *gateway tracker*. C uses G_C for all its future requests into the network. If at any point G_C fails, C repeats this entire process in order to find a new gateway tracker.

KeepAlive()	Tracker $\rightarrow$ Tracker	Used by the tracker network to check tracker liveness.
InitPred/InitSucc$()$	Tracker $\rightarrow$ Tracker	Sent to a tracker's successor and predecessor when entering the tracker network.
InitRouteTable$()$	Tracker $\rightarrow$ Tracker	Used to initialize a tracker's routing table when entering the tracker network.
Update$(ip, port, type)$	Tracker $\rightarrow$ Tracker	Used to send updates between a subset of nodes.
Event$(ip, port, type)$	Tracker $\rightarrow$ Tracker	Used to send events to a slice leader.
FindTracker$(content\text{-}id)$	Client $\rightarrow$ Tracker	Used by a client to request the $node\text{-}ids$ of nodes tracking $content\text{-}id$.
IsTracking$(track)$	Tracker $\rightarrow$ Tracker	Used to probe potential file trackers to determine whether they are already or are willing to track a particular file.
SynchronizeTrack$(track)$	Tracker $\rightarrow$ Tracker	Used to synchronize $track$ metadata between co-trackers. $track$ contains a list of clients and the status of their transfers.
NotifyRetirement$(track)$	Tracker $\rightarrow$ Tracker	Used by a tracker to notify co-tracker peers that it is no longer tracking a given file.

Table 1. Messages used by eTorrent.

In order to increase the chances of C finding some active tracker, we update and maintain a known tracker list much like in Gnutella [6]. Every time C finds a new gateway tracker, it updates its known trackers list with a subset of the G_C's global routing table.

3.2. Downloading Files

To download a file, client C first acquires the ContentId(f) of the interested file through some external search. Next, C contacts its gateway tracker G_C asking for a tracker that is responsible for tracking f. G_C returns to C a random subset of trackers that are tracking the file. C then attempts to contact some T_f from this subset; if it fails, it asks G_C for a new tracker until it finds one that accepts its request to download. Note that a T_f may reject a client based on its current load, even if is currently tracking that particular file. Once C successfully finds a tracker it initiates a download session via conventional BitTorrent processes.

3.3. Seeding Files

When C wants to share some file f, it first calculates $\mathsf{ContentId}(f)$. It then contacts the tracker network which returns a list of potential trackers that can potentially track f given the constraints in section 2.2. It is the responsibility of the client to contact nodes on this list to find a working tracker.

Once a tracker T_f is found, C contacts T_f via normal BitTorrent protocol. However, instead of reporting that it started a download, it reports that it has a completed download. Trackers do not explicitly differentiate clients that are seeds from those that are leechers.

3.4. Tracker Replication

The tracker network tries to maintain at least k replicated trackers for each file. When T_f first encounters a new file (when a client seeds the file), it creates a new track associated with this file. It then proceeds to find $k - 1$ co-trackers to maintain this track.

The replicated tracker set (or *co-tracker* list) is established by probing tracker nodes starting at $\mathsf{succ}(\mathsf{LocateId}(f, 0))$. For each node, the T_f sends a $\mathsf{IsTracking}(f)$ message. The given node may reply true, false, or unwilling. When the node is unwilling, then it is abnegating responsibility because of bandwidth or load limitations. T_f stops probing when it has found $k - 1$ co-trackers.

Once a co-tracker set has been established, T_f will regularly synchronizes track information with nodes in the set. Synchronization of track information ensures that a client can connect to any of the k trackers, and receive the same client list. Clients may not get a consistent client list all the time, but over time, the list will eventually become consistent.

During synchronization each tracker broadcasts its current client list to all co-trackers. This update will contain a list of clients with ownership information and the BitTorrent client metadata. A client is defined as *owned* by a tracker if it has made direct contact with that tracker.

Each tracker maintains a table of all the updates it received from its replicated trackers. Each track is updated by taking the union of the received client lists to create a complete view of clients.

During the merging of client lists, there may be conflicting information about clients that have multiple tracker owners. This can occur if the client contacted several different trackers. To resolve potential merge conflicts, the owner which reports the most download progress (least bytes left to download) for a particular client will be assigned as that client's primary owner. Only track information from the primary owner copy will be used in the merged list.

3.5. Tracker Introduction

When a new tracker T enters the system, it first contacts the tracker network. Through the one-hop protocol, T learns of its successor, predecessor and the global routing table. In addition, the predecessor and successor report T's arrival to their slice leader which eventually propagates this event to every other node in the system.

An entering tracker T may be responsible for tracking files already existing in the network. After entering the network, f's existing co-trackers will broadcast their track information to T, making it a new co-tracker for f.

3.6. Tracker Failure

Every so often, a keep-alive message is sent from each tracker node to its successor. If a node learns that its successor has failed, it immediately reports this to its leader which eventually propagates via the one-hop protocol to every other node.

Once all T_f nodes have been notified of the failure event, f's existing co-trackers will find a new tracker and send their tracks to it.

4. Implementation

eTorrent is implemented in Python so as to smoothly integrate with the original BitTorrent client and tracker.

4.1. Client

BitTorrent already provides functionality for performing a swarm download given a hash and tracker list in a `.torrent` file. In BitTorrent, the `.torrent` file contains an "announce" URL that points to the tracker. In the eTorrent system, the announce URL is ignored, and announces URLs are generated dynamically from the tracker list provided by the gateway G_C. We implemented a usable eTorrent client that wraps around an existing BitTorrent client.

The user provides a content identifier either directly (from user input) or through a pre-existing `.torrent` file. The client C next sends this content identifier to G_C, which in turn returns a set of trackers to C. The wrapper client generates an "announce" URL corresponding to one of the trackers in the returned list and hands this to the standard BitTorrent client for further processing.

The eTorrent client periodically checks in with its tracker via the "announce" URL, announcing its progress during the download. If it fails to connect to the tracker, the client will locate another tracker either from its list or by contacting the tracker network.

4.2. Tracker

Given the tracker-file mapping described in section 2.2, we desire identifiers to be close to uniformly distributed given some random set of files that are on the network. In addition, in order to make it difficult for trackers to enter wherever they wish, the hash should be noninvertible. This is important because otherwise a malicious party could easily block access to some file f by placing itself as $\mathsf{succ}(\mathsf{LocateId}(f, s))$ and thus becoming the trackers of f. In order to achieve these properties, we use a 160-bit SHA1 hash to calculate all the identifiers.

In our eTorrent tracker we use a threaded-model with locks to secure the tracker's internal shared state. Four threads were spawned for each tracker: one for handling client connections, one for one-hop routing, one for co-tracker synchronization and one for eTorrent message handling. The first thread runs the original BitTorrent tracker; the next three will be discussed in more detail below.

4.3. One-Hop Routing

The main part of one-hop routing's implementation is described in [11]. We choose for simplicity to elect slice leaders as the successor of the median of the slice. The number of slices and units in the system can be either statically fixed or dynamically chosen. In the latter case, the number of slice and units is increased or decreased if the average number of nodes per unit hits certain thresholds.

Nodes use three different type of notifications: internal events, which are events that occur within a slice but has not been propagated to the whole network; external events, which are events that have been propagated to the whole network; and updates, which are external events that have been propagated from slice leaders to their corresponding nodes.

Note that one-hop routing itself does not guarantee that the global routing tables will stay consistent. Events/updates can be lost, or timing issues may cause updates to not be sent to particular nodes. To combat this, if a tracker notices an inconsistency in a routing table, it will propagate a new event corresponding to this error. For example, if a tracker T receives a track update from another tracker or query from a client regarding some file f that it is not tracking, this implies that there is a discrepancy regarding $\mathsf{pred}(T)$. Thus T will check its predecessor's status and send out the corresponding event. Clients also notify their gateway trackers G_C if they notice some tracker is down; the gateway tracker will independently verify this and propagate this event if necessary.

4.4. Co-Tracker Synchronization

During track synchronization trackers use the global routing table to determine the status of co-trackers. As mentioned earlier, trackers responsible for tracking f will each independently maintain a list of k co-trackers (where k is the network minimum on the number of co-trackers per track).

The co-trackers list is cached, so that each tracker can avoid probing its co-trackers every synchronizing cycle. This cached list is checked each cycle against the one-hop routing table to prune away any dead tracker nodes in the list. When the size of the cached list falls below k, the co-tracker list is rebuilt.

During every synchronization cycle, each tracker sends its client information via the $\mathsf{SynchronizeTrack}(f)$ message. All the client metadata is encoded using the same binary encoding format that BitTorrent uses for its network protocol. Removal of expired clients is automatically taken care of either when a client announces that its disconnecting or when it fails to report after the required interval.

We realize that a full client list update is expensive when the number of clients per track grows. We plan to extend the synchronization protocol to allow for partial updates.

4.5. Tracker Retirement

Occasionally, when a new tracker joins or leaves the network, the responsibility of a file tracker may change. For example, if some tracker $T_f = \mathsf{succ}(\mathsf{LocateId}(\text{f,k}))$ and a new tracker T' enters the system such that $\mathsf{LocateId}(f, k) < \mathsf{NodeId}(T') < \mathsf{NodeId}(T_f)$, T_f will *retire* its responsibilities to T' by sending it a full list of clients. If the full update completes successfully, the retiring tracker will also notify its former co-trackers of this event through the $\mathsf{NotifyRetirement}(f)$ message. This mechanism ensures that subsequent updates by peer co-trackers will route correctly, and that new clients can still find the file trackers.

When a tracker retires, all clients connecting to that tracker must shift to the new tracker. Ideally the retired tracker should be up long enough so that all its clients can shift to the new tracker. So, in addition to the minimum k trackers per file, we also set a global constant $m > k$ that specifies the maximum number of trackers per file f. This constant allows the network to buffer against quick network membership changes. A retired tracker may stay on tracking that file, as long as the number of total trackers remain below m.

4.6. Communication

Nodes communicate in the network via a Remote Procedure Call (RPC) mechanism. We leverage the XML-RPC library for Python [12] in order to achieve this. We implemented all the messages listed in Table 1 using this RPC package. It should be noted that messages in the original BitTorrent protocol are not handled by this method; messages between the original BitTorrent client and tracker remain unchanged.

5. System Performance

To test the performance of eTorrent network we created a series of test networks with 100 trackers, split into 2 slices and 4 units. We gathered bandwidth results by using Ethereal and performed a TCP dump on the appropriate tracker ports on all systems running trackers. In this setup, the co-tracker synchronization occurs once a minute, and one-hop keep-alives are sent once per second. For our analyses we further assume that nodes which enter the network have an average lifetime of one hour and that the average message transmission latency is 100 ms.

5.1. Global Routing Table Divergence

From our preliminary network analysis, we noticed that discrepancies occurred in the global routing table frequently between nodes in the network. Keeping the global routing table consistent is a central part of our system. Through further analysis, we narrowed down the source of discrepancies to two causes:

1. When a node passing a value fails immediately after receiving an update message but before it can propagate the update to the next node, all nodes that require updates from the failed node will experience a discrepancy in their global routing table.

2. If a new node were to enter a new unit (or slice) and become its leader, its slice leader (or other slice leaders) do not know which of the recent updates the new leader has received. Thus it could potentially miss several updates.

These inconsistencies can be found by tracker or client during routine operations. Once detected these discrepancies can be easily fixed by the network as described in section 4.3.

To quantify the possible error rate due to these two sources. we derived an equation for the probability of error using results from the one-hop routing paper [11]:

$$P_{err} = E^2(U+3) + E^2\frac{4}{U} + E^2\frac{2(S+U)}{SU}$$

Here E is the expected number of entries (or exits) during a time step, U is the number of nodes in a unit, and S is the number of units in a slice. The first term represents error due to failing nodes, and the second and third terms represent errors due to unit and slice number changes respectively. With the assumed values for our test scenario, $P_{err} = 7.33 \cdot 10^{-9}$ which equates to approximately 0.0063 errors per day.

We ran further experiments to study the actual effects of node changes (trackers entering and exiting the network) and its effect on the overall error reported in the global routing tables. For a given *churn rate* r, we started off a network by having 100 random nodes join at a rate of r. We then simulated node changes by randomly killing and adding new trackers at the same churn rate. After performing 50 changes and allowing the one-hop routing tables to reach a steady state, we logged each node's global routing table and compared it against a correct global routing table, noting any discrepancies we found.

We ran this test five times for each r we used and averaged the results. Note that the error correction described in section 4.3 was disabled, since it relies on the file track lookups by a client or co-tracker, neither of which were performed for these tests.

Figure 2 shows the average of discrepancies measure varying churn rate from 0.5 changes per second (2 seconds between changes) to 100 changes per second (0.01 seconds between changes). Our results show that even with frequent changes in the network (100 changes per second) the average node experiences only 2.1 differences from the true global routing table.

Figure 3 shows the average number of unique discrepancies versus membership change in the system. We can see that the number of unique errors has a sublinear (roughly logarithmic) growth rate. This implies that when corrective measures are taken in networks with higher churn rates (less

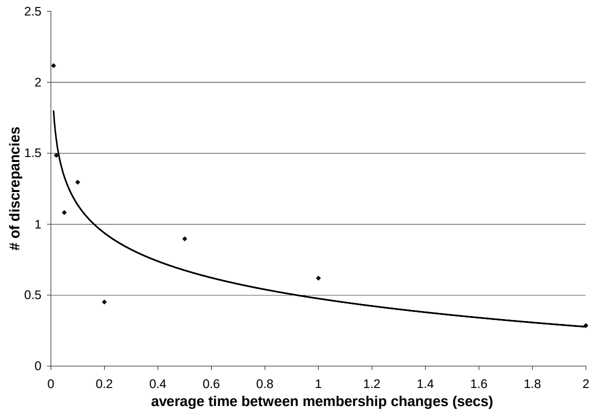

Figure 2. Average number of errors per node versus time between node changes

time between changes), each correction will have a larger effect on the total percentage of errors in the system. This makes sense as the errors due to one-hop routing tend to have high locality.

Our experimental results show that in general one-hop routing will produce fairly consistent routing tables. Without correction the discrepancy rate is fairly reasonable. Further improvements can be achieved with the simple error correction described in section 4.3. Therefore, we expect that our scheme will scale for reasonably large P2P networks.

5.2. Bandwidth Overhead

From analysis of TCP dump from Ethereal, we were able to measure the network overhead of various messages in our system. We found that the cost of synchronization of the network tables via the one-hop system took 350 bytes per second. This amount remains consistent as we vary network size.

The current file tracker synchronization (with full updates) requires roughly 45 bytes per second per file tracked with an additional 1 byte per second per client for that file. This may increase for very popular files (which has larger client lists). We believe that by modifying the synchronization methods to perform differential updates, we can drasti-

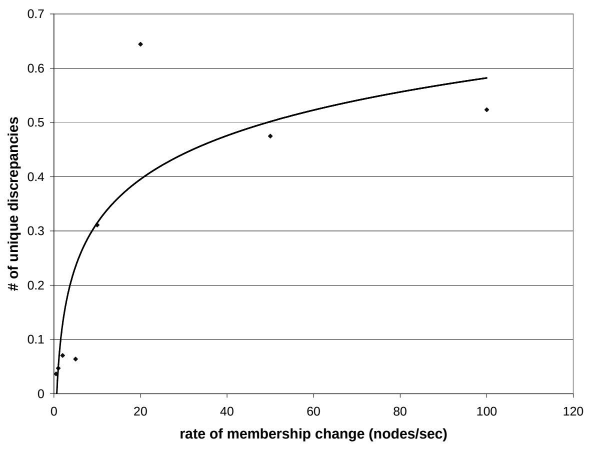

Figure 3. Average number of unique errors per node versus time between node changes

cally cut down these costs.

A major overhead for communication was the marshalling method of XML-RPC. A simple compressed version of it would yield a factor of two improvement. Other reductions in the format could further reduce this by an additional 50%.

6. Future Work

While meeting the initial goal of distributing and replicating the task of file tracking seamlessly, there are a number of problems that prevent the system from widespread use. In this section we discuss these problems in eTorrent and possible solutions.

6.1. Tracker Rewards

Currently, there is no motivation for someone to volunteer to be a tracker node in the eTorrent network. Because we want the responsibility of tracking a file to be shared among clients and not dedicated servers, clients would ideally act as trackers when connected to the system. However, tracking a file adds load to a node system, as it has to potentially communicate with many trackers and clients.

In order to promote volunteers to track, a reward system could be added that to compensate trackers for providing tracking services. One proposed method would act as follows:

1. Client C maintains a list of trackers it has used in the past, rewarding them one credit for each track used.
2. When some client T is interested in file pieces that C has, and T has served as C's tracker in the past, C will favor uploading content to T over other nodes and deduct one credit.
3. This list will be maintained using an LRU policy; credits can expire if not used.

The mechanism to choose whom to upload content is already an integral part of the BitTorrent algorithm, so adding this extra condition of which client to favor would be relatively simple. This system would also be difficult to cheat, as each client keeps track of which nodes should receive benefits.

6.2. Security

The current system assumes that trackers are trustworthy (i.e. they do not attempt to sabotage the network). With the current method of getting assigned IDs based on SHA1, it is difficult, but computationally reasonable, to situate malicious nodes such that they are assigned to be file trackers for certain files. Additional work will be needed to prevent nodes from disrupting the routing tables.

6.3. Routing Robustness

Optimizations in the global routing table synchronization protocol are possible. The election of slice and unit leaders in the one-hop routing system could be done only when needed (i.e. if the leader went down). Additionally, trackers could be entered into the system from the pool of possible trackers (i.e. clients) as needed instead of the aggressive system.

6.4. File Tracking Scaling

A quality-of-service measure should be implemented to vary the number of trackers per file based on load. Such a feature could also be used to do more sophisticated load-balancing amongst the trackers. This would allow the system to efficiently allocate more co-trackers to the files that need it most.

6.5. Communication

Using the XML-RPC library incurred a large penalty. Analysis revealed that RPC calls used roughly 300% the message space to marshall calls over what could have been accomplished with a more ad-hoc protocol. Such a change would greatly extend the scalability of the eTorrent system.

6.6. Searching

The eTorrent system currently has no integrated way of searching for content on the system. Proposals for performing a general distributed meta-data search have been discussed [9]. It would be straightforward to merge these systems in order to incorporate this functionality.

7 Conclusion

The eTorrent network we designed successfully distributes the task of tracking files across multiple nodes to provide robustness in the face of tracker failure or overloading with an acceptable bandwidth overhead. Although there are still limitations in the system that prevent it from being a viable choice as a general purpose P2P system, there are clear steps to take in order to make the eTorrent system a feasible real-life application.

Acknowledgements

We would like to thank Robert Morris for advice and guidance in determining the scope of this work.

References

[1] Anon. BitTorrent Protocol Specification. http://wiki.theory.org/index.php/BitTorrentSpecification, March 2004.

[2] B. Cohen. BitTorrent Specification v1.0. http://wiki.theory.org/index.php/BitTorrentSpecification.

[3] J. Hoffman. Multitracker Metadata Entry Specification. http://bittornado.com/docs/multitracker-spec.txt, March 2004.

[4] B. Cohen. Incentives Build Robustness in BitTorrent. http://bitconjurer.org/BitTorrent/bittorrentecon.pdf, May 2003.

[5] Anon. Shareaza FAQ on BitTorrent. http://shareazawiki.anenga.com/tiki-index.php?page=FAQ:BitTorrent

[6] Anon. The Gnutella Protocol Specification v0.41. http://www.limewire.com/developer/gnutella_protocol_0.4.pdf.

[7] eDonkey2000. http://www.edonkey.com.

[8] A. Metz and C. Gieseler. Conglomeration and Search of BitTorrent Content Data. http://www.cs.iastate.edu/~charlesg/ie574/ie574Proposal.pdf, January 2004.

[9] S. Joseph and T. Hoshiai. Decentralized Meta-Data Strategies: Effective Peer-to-Peer Search. In *IEICE Transactions on Communications*, Vol.E86-B No.6, pages 1740-1753.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. Frans Kaashoek, F. Dabek, H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*. http://www.pdos.lcs.mit.edu/chord/papers/paper-ton.pdf.

[11] A. Gupta, B. Liskov and R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. In *Proc. HotOS-IX*, Kauai, HI, May 2003.

[12] UserLand Software, Inc. http://www.xmlrpc.com.