

# apt-p2p: A Peer-to-Peer Distribution System for Software Package Releases and Updates

Cameron Dale  
School of Computing Science  
Simon Fraser University  
Burnaby, British Columbia, Canada  
Email: camerond@cs.sfu.ca

Jiangchuan Liu  
School of Computing Science  
Simon Fraser University  
Burnaby, British Columbia, Canada  
Email: jcliu@cs.sfu.ca

**Abstract**—The Internet has become a cost-effective vehicle for software development and release, particular in the free software community. Given the free nature of this software, there are often a number of users motivated by altruism to help out with the distribution, so as to promote the healthy development of this voluntary society. It is thus naturally expected that a peer-to-peer distribution can be implemented, which will scale well with large user bases, and can easily explore the network resources made available by the volunteers.

Unfortunately, this application scenario has many unique characteristics, which make a straightforward adoption of existing peer-to-peer systems for file sharing (such as BitTorrent) suboptimal. In particular, a software release often consists of a large number of packages, which are difficult to distribute individually, but the archive is too large to be distributed in its entirety. The packages are also being constantly updated by the loosely-managed developers, and the interest in a particular version of a package can be very limited depending on the computer platforms and operating systems used.

In this paper, we propose a novel peer-to-peer assisted distribution system design that addresses the above challenges. It enhances the existing distribution systems by providing compatible and yet more efficient downloading and updating services for software packages. Our design leads to apt-p2p, a practical implementation that extends the popular apt distributor. apt-p2p has been used in conjunction with Debian-based distribution of Linux software packages and is also available in the latest release of Ubuntu. We have addressed the key design issues in apt-p2p, including indexing table customization, response time reduction, and multi-value extension. They together ensure that the altruistic users' resources are effectively utilized and thus significantly reduces the currently large bandwidth requirements of hosting the software, as confirmed by our existing real user statistics gathered over the Internet.

## I. INTRODUCTION

With the widespread penetration of broadband access, the Internet has become a cost-effective vehicle for software development and release [1]. This is particularly true for the free software community whose developers and users are distributed worldwide and work asynchronously. The ever increasing power of modern programming languages, computer platforms, and operating systems has made this software extremely large and complex, though it is often divided into a huge number of small packages. Together with their popularity among users, an efficient and reliable management and distribution of these software packages over the Internet has become a challenging task [2].

The existing distribution for free software is mostly based on the client/server model, e.g., the Advanced Package Tool (apt) for Linux [3], which suffers from the well-known bottleneck problem. Given the free nature of this software, there are often a number of users motivated by altruism to help out with the distribution, so as to promote the healthy development of this voluntary society. We thus naturally expect that peer-to-peer distribution can be implemented in this context, which will scale well with the currently large user bases and can easily explore the resources made available by the volunteers.

Unfortunately, this application scenario has many unique characteristics, which make a straightforward adoption of existing peer-to-peer systems for file sharing (such as BitTorrent) suboptimal. In particular, there are too many packages to distribute each individually, but the archive is too large to distribute in its entirety. The packages are also being constantly updated by the loosely-managed developers, and the interest in a particular version of a package can be very limited. They together make it very difficult to efficiently create and manage torrents and trackers. The random downloading nature of BitTorrent-like systems is also different from the sequential order used in existing software package distributors. This in turn suppresses interaction with users given the difficulty in tracking speed and downloading progress.

In this paper, we propose a novel peer-to-peer assisted distribution system design that addresses the above challenges. It enhances the existing distribution systems by providing compatible and yet more efficient downloading and updating services for software packages. Our design leads to the development of apt-p2p, a practical implementation based on the Debian<sup>1</sup> package distribution system. We have addressed the key design issues in apt-p2p, including indexing table customization, response time reduction, and multi-value extension. They together ensure that the altruistic users' resources are effectively utilized and thus significantly reduces the currently large bandwidth requirements of hosting the software.

apt-p2p has been used in conjunction with the Debian-based distribution of Linux software packages and is also available in the latest release of Ubuntu. We have evaluated our current deployment to determine how effective it is at

<sup>1</sup>Debian - The Universal Operating System: <http://www.debian.org/>

meeting our goals, and to see what effect it is having on the Debian package distribution system. In particular, our existing real user statistics have suggested that it responsively interacts with clients and substantially reduces server cost.

The rest of this paper is organized as follows. The background and motivation are presented in Section II, including an analysis of BitTorrent’s use for this purpose in Section II-C. We propose our solution in Section III. We then detail our sample implementation for Debian-based distributions in Section IV, including an in-depth look at our system optimization in Section V. The performance of our implementation is evaluated in Section VI. We examine some related work in Section VII, and then Section VIII concludes the paper and offers some future directions.

## II. BACKGROUND AND MOTIVATION

In the free software community, there are a large number of groups using the Internet to collaboratively develop and release their software. Efficient and reliable management and distribution of these software packages over the Internet thus has become a critical task. In this section, we offer concrete examples illustrating the unique challenges in this context.

### A. Free Software Package Distributors

Most Linux distributions use a software package management system that fetches packages to be installed from an archive of packages hosted on a network of mirrors. The Debian project, and other Debian-based distributions such as Ubuntu and Knoppix, use the `apt` (Advanced Package Tool) program, which downloads Debian packages in the `.deb` format from one of many HTTP mirrors. The program will first download index files that contain a listing of which packages are available, as well as important information such as their size, location, and a hash of their content. The user can then select which packages to install or upgrade, and `apt` will download and verify them before installing them.

There are also several similar frontends for the RPM-based distributions. Red Hat’s Fedora project uses the `yum` program, SUSE uses `YAST`, while Mandriva has `Rpmdrake`, all of which are used to obtain RPMs from mirrors. Other distributions use tarballs (`.tar.gz` or `.tar.bz2`) to contain their packages. Gentoo’s package manager is called `portage`, SlackWare Linux uses `pkgtools`, and FreeBSD has a suite of command-line tools, all of which download these tarballs from web servers.

Similar tools have been used for other types of software packages. CPAN distributes packaged software for the PERL programming language, using SOAP RPC requests to find and download files. Cygwin provides many of the standard Unix/Linux tools in a Windows environment, using a package management tool that requests packages from websites. There are two software distribution systems for software that runs on the Macintosh OS, `fink` and `MacPorts`, that also retrieve packages in this way.

Direct web downloading by users is also common, often coupled with a hash verification file to be downloaded next to

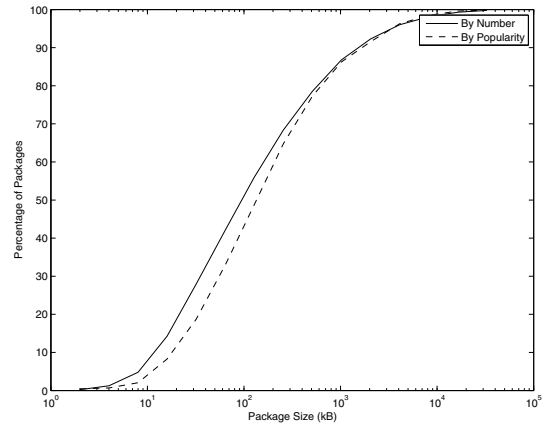


Fig. 1. The CDF of the size of packages in a Debian system, both for the actual size and adjusted size based on the popularity of the package.

the desired file. The hash file usually has the same file name, but with an added extension identifying the hash used (e.g. `.md5` for the MD5 hash). This type of file downloading and verification is typical of free software hosting facilities that are open to anyone to use, such as SourceForge.

Given the free nature of this software, there are often a number of users motivated by altruism to want to help out with the distribution. This is particularly true considering that much of this software is used by groups that are staffed mostly, or sometimes completely, by volunteers. They are thus motivated to contribute their network resources, so as to promote the healthy development of the volunteer community that released the software. We also naturally expect that peer-to-peer distribution can be implemented in this context, which will scale well with the currently large user bases and can easily explore the network resources made available by the volunteers.

### B. Unique Characteristics

While it seems straightforward to use an existing peer-to-peer file sharing tool like BitTorrent for this free software package distribution, there are indeed a series of new challenges in this unique scenario:

1) *Archive Dimensions*: While most of the packages of a software release are very small in size, there are some that are quite large. There are too many packages to distribute each individually, but the archive is also too large to distribute in its entirety. In some archives there are also divisions of the archive into sections, e.g. by the operating system (OS) or computer architecture that the package is intended for.

For example, Figure 1 shows the size of the packages in the current Debian distribution. While 80% of the packages are less than 512 KB, some of the packages are hundreds of megabytes. The entire archive consists of 22,298 packages and is approximately 119,000 MB in size. Many of the packages are to be installed in any computer environment, but there are also OS- or architecture-specific packages.



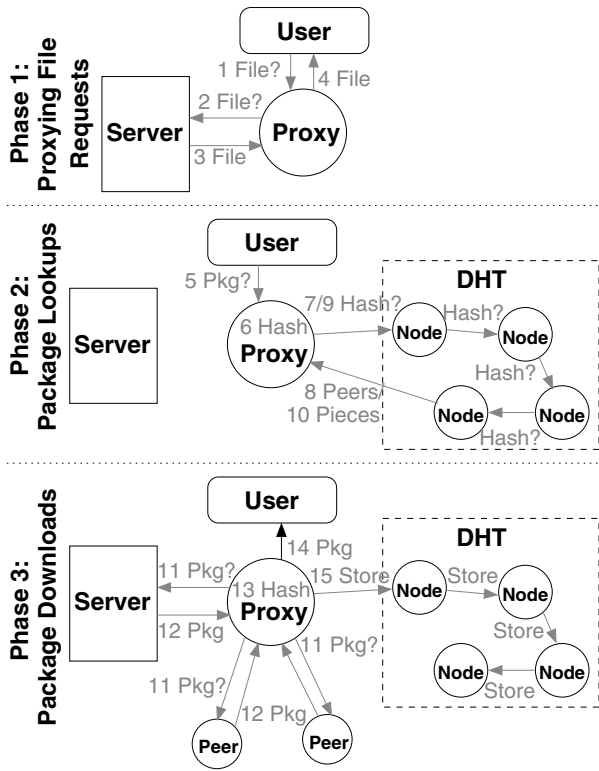


Fig. 4. The different phases of functionality of our peer-to-peer distribution model.

tial downloads.

On the other hand, there are aspects of BitTorrent that are no longer critical. Specifically, with altruistic peers and all files being available to download without uploading, incentives to share become a less important issue. Also, the availability of seeders is not critical either, as the servers are already available to serve in that capacity.

### III. PEER-TO-PEER ASSISTED DISTRIBUTOR: AN OVERVIEW

We now present the design of our peer-to-peer assisted distribution system for free software package releases and updates. A key principle in our design is that the new functionalities implemented in our distributor should be transparent to users, thus offering the same experience as using conventional software management systems, but with enhanced efficiency. That said, we assume that the user is still attempting to download packages from a server, but the requests will be proxied by our peer-to-peer program. We further assume that the server is always available and has all of the package files. In addition, the cryptographic hash of the packages will be available separately from the package itself, and is usually contained in an *index* file which also contains all the packages' names, locations and sizes.

#### A. System Overview

Our model for using peer-to-peer to enhance package distribution is shown in Figure 4. As shown in Phase 1, our program

will act as a proxy (1,2), downloading (3) and caching all files communicated between the user and the server (4). It will therefore also have available the index files containing the cryptographic hashes of all packages. Later, in Phase 2, upon receiving a request from the user to download a package (5), our program will search the index files for the package being requested and find its hash (6). This hash can then be looked up recursively in an indexing structure (a Distributed Hash Table, or DHT [7], in our implementation) (7), which will return a list of peers that have the package already (8). Then, in Phase 3, the package can be downloaded from the peers (11,12), it can be verified using the hash (13), and if valid can be returned to the user (14). The current node's location is also added to the DHT for that hash (15), as it is now a source for others to download from.

In steps (11,12), the fact that this package is also available to download for free from a server is very important to our proposed model. If the package hash can not be found in the DHT, the peer can then fallback to downloading from the original location (i.e. the server). The server thus, with no modification to its functionality, serves as a seed for the packages in the peer-to-peer system. Any packages that have just been updated or that are very rare, and so do not yet have any peers available, can always be found on the server. Once the peer has completed the download from the server and verified the package, it can then add itself to the DHT as the first peer for the new package, so that future requests for the package will not need to use the server.

This sparse interest in a large number of packages undergoing constant updating is well suited to the functionality provided by a DHT. A DHT requires unique keys to store and retrieve strings of data, for which the cryptographic hashes used by these package management systems are perfect for. The stored and retrieved strings can then be pointers to the peers that have the package that hashes to that key.

Note that, despite downloading the package from untrustworthy peers, the trust of the package is always guaranteed through the use of the cryptographic hashes. Nothing can be downloaded from a peer until the hash is looked up in the DHT, so a hash must first come from a trusted source (i.e. the distributor's server). Most distributors use index files that contain hashes for a large number of the packages in their archive, and which are also hashed. After retrieving the index file's hash from the server, the index file can also be downloaded from peers and verified. Then the program has access to all the hashes of the packages it will be downloading, all of which can be verified with a *chain of trust* that stretches back to the original distributor's server.

#### B. Peer Downloading Protocol

Although not necessary, we recommend implementing a download protocol that is similar to the protocol used to fetch packages from the distributor's server. This simplifies the peer-to-peer program, as it can then treat peers and the server almost identically when requesting packages. In fact, the server can

be used when there are only a few slow peers available for a file to help speed up the download process.

Downloading a file efficiently from a number of peers is where BitTorrent shines as a peer-to-peer application. Its method of breaking up larger files into pieces, each with its own hash, makes it very easy to parallelize the downloading process and maximize the download speed. For very small packages (i.e. less than the piece size), this parallel downloading is not necessary, or even desirable. However, this method should still be used, in conjunction with the DHT, for the larger packages that are available.

Since the package management system only stores a hash of the entire package, and not of pieces of that package, we will need to be able to store and retrieve these piece hashes using the peer-to-peer protocol. In addition to storing the file download location in the DHT (which would still be used for small files), a peer will store a *torrent string* containing the peer's hashes of the pieces of the larger files (similar to (15) in Phase 3 of Figure 4). These piece hashes will be retrieved and compared ahead of time by the downloading peer ((9,10) in Phase 2 of Figure 4) to determine which peers have the same piece hashes (they all should), and then used during the download to verify the pieces of the downloaded package.

#### IV. APT-P2P: A PRACTICAL IMPLEMENTATION

We have created a sample implementation that functions as described in section III, and is freely available for other distributors to download and modify [8]. This software, called `apt-p2p`, interacts with the popular `apt` tool. This tool is found in most Debian-based Linux distributions, with related statistics available for analyzing the popularity of the software packages [5].

Since all requests from `apt` are in the form of HTTP downloads from a server, our implementation takes the form of a caching HTTP proxy. Making a standard `apt` implementation use the proxy is then as simple as prepending the proxy location and port to the front of the mirror name in `apt`'s configuration file (i.e. "`http://localhost:9977/mirrorname.debian.org/...`").

We created a customized DHT based on Khashmir [9], which is an implementation of Kademia [7]. Khashmir is also the same DHT implementation used by most of the existing BitTorrent clients to implement trackerless operation. The communication is all handled by UDP messages, and RPC (remote procedure call) requests and responses between nodes are all *encoded* in the same way as BitTorrent's `.torrent` files. More details of this customized DHT can be found below in Section V.

Downloading is accomplished by sending simple HTTP requests to the peers identified by lookups in the DHT to have the desired file. Requests for a package are made using the package's hash (properly encoded) as the URL to request from the peer. The HTTP server used for the proxy also doubles as the server listening for requests for downloads from other peers. All peers support HTTP/1.1, both in the server and the client, which allows for pipelining of multiple requests to a

peer, and the requesting of smaller pieces of a large file using the HTTP Range request header. Like in `apt`, SHA1 hashes are then used to verify downloaded files, including the large index files that contain the hashes of the individual packages.

#### V. SYSTEM OPTIMIZATION

Another contribution of our work is in the customization and use of a Distributed Hash Table (DHT). Although our DHT is based on Kademia, we have made many improvements to it to make it suitable for this application. In addition to a novel storage technique to support piece hashes, we have improved the response time of looking up queries, allowed the storage of multiple values for each key, and incorporated some improvements from BitTorrent's tracker-less DHT implementation.

##### A. DHT Details

DHTs operate by storing (*key, value*) pairs in a distributed fashion such that no node will, on average, store more or have to work harder than any other node. They support two primitive operations: *put*, which takes a key and a value and stores it in the DHT; and *get*, which takes a key and returns a value (or values) that was previously stored with that key. These operations are recursive, as each node does not know about all the other nodes in the DHT, and so must recursively search for the correct node to put to or get from.

The Kademia DHT, like most other DHTs, assigns Ids to peers randomly from the same space that is used for keys. The peers with Ids closest to the desired key will then store the values for that key. Nodes support four primitive requests. `ping` will cause a peer to return nothing, and is only used to determine if a node is still alive. `store` tells a node to store a value associated with a given key. The most important primitives are `find_node` and `find_value`, which both function recursively to find nodes close to a key. The queried nodes will return a list of the nodes they know about that are closest to the key, allowing the querying node to quickly traverse the DHT to find the nodes closest to the desired key. The only difference between `find_node` and `find_value` is that the `find_value` query will cause a node to return a value, if it has one for that key, instead of a list of nodes.

##### B. Piece Hash Storage

Hashes of pieces of the larger package files are needed to support their efficient downloading from multiple peers. For large files (5 or more pieces), the torrent strings described in Section III-B are too long to store with the peer's download info in the DHT. This is due to the limitation that a single UDP packet should be less than 1472 bytes to avoid fragmentation.

Instead, the peers will store the torrent string for large files separately in the DHT, and only contain a reference to it in their stored value for the hash of the file. The reference is an SHA1 hash of the entire concatenated length of the torrent string. If the torrent string is short enough to store separately in the DHT (i.e. less than 1472 bytes, or about 70 pieces for the SHA1 hash), then a lookup of that hash in the DHT will return the torrent string. Otherwise, a request to the peer for

the hash (using the same method as file downloads, i.e. HTTP), will cause the peer to return the torrent string.

Figure 1 shows the size of the 22,298 packages available in Debian in January 2008. We can see that most of the packages are quite small, and so most will therefore not require piece hash information to download. We have chosen a piece size of 512 kB, which means that 17,515 (78%) of the packages will not require this information. There are 3054 packages that will require 2 to 4 pieces, for which the torrent string can be stored directly with the package hash in the DHT. There are 1667 packages that will require a separate lookup in the DHT for the longer torrent string, as they require 5 to 70 pieces. Finally, there are only 62 packages that require more than 70 pieces, and so will require a separate request to a peer for the torrent string.

### C. Response Time Optimization

Many of our customizations to the DHT have been to try and improve the time of the recursive `find_value` requests, as this can cause long delays for the user waiting for a package download to begin. The one problem that slows down such requests is waiting for timeouts to occur before marking the node as failed and moving on.

Our first improvement is to retransmit a request multiple times before a timeout occurs, in case the original request or its response was lost by the unreliable UDP protocol. If it does not receive a response, the requesting node will retransmit the request after a short delay. This delay will increase exponentially for later retransmissions, should the request again fail. Our current implementation will retransmit the request after 2 seconds and 6 seconds (4 seconds after the first retransmission), and then timeout after 9 seconds.

We have also added some improvements to the recursive `find_node` and `find_value` queries to speed up the process when nodes fail. If enough nodes have responded to the current query such that there are many new nodes to query that are closer to the desired key, then a stalled request to a node further away will be dropped in favor of a new request to a closer node. This has the effect of leap-frogging unresponsive nodes and focussing attention on the closer nodes that do respond. We will also prematurely abort a query while there are still outstanding requests, if enough of the closest nodes have responded and there are no closer nodes found. This prevents a far away unresponsive node from making the query's completion wait for it to timeout.

Finally, we made all attempts possible to prevent firewalled and NATted nodes from being added to the routing table for future requests. Only a node that has responded to a request from us will be added to the table. If a node has only sent us a request, we attempt to send a `ping` to the node to determine if it is NATted or not. Unfortunately, due to the delays used by NATs in allowing UDP packets for a short time if one was recently sent by the NATted host, the ping is likely to succeed even if the node is NATted. We therefore also schedule a future ping to the node to make sure it is still reachable after the NATs delay has hopefully elapsed. We also schedule future

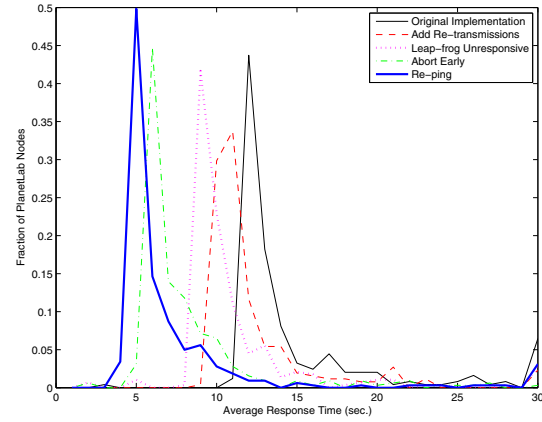


Fig. 5. The distribution of average response times PlanetLab nodes experience for `find_value` queries. The original DHT implementation results are shown, as well as the successive improvements that we made to reduce the response time.

pings of nodes that fail once to respond to a request, as it takes multiple failures (currently 3) before a node is removed from the routing table.

To test our changes during development, we ran our customized DHT for several hours after each major change on over 300 PlanetLab nodes [10]. Though the nodes are not expected to be firewalled or NATted, some can be quite overloaded and so consistently fail to respond within a timeout period, similar to NATted peers. The resulting distribution of the nodes' average response times is shown in Figure 5. Each improvement successfully reduced the response time, for a total reduction of more than 50%. The final distribution is also narrower, as the improvements make the system more predictable. However, there are still a large number of outliers with higher average response times, which are the overloaded nodes on PlanetLab. This was confirmed by examining the average time it took for a timeout to occur, which should be constant as it is a configuration option, but can be much larger if the node is too overloaded for the program to be able to check for a timeout very often.

### D. Multiple Values Extension

The original design of Kademlia specified that each key would have only a single value associated with it. The RPC to find this value was called `find_value` and worked similarly to `find_node`, iteratively finding nodes with Id's closer to the desired key. However, if a node had a value stored associated with the searched for key, it would respond to the request with that value instead of the list of nodes it knows about that are closer.

While this works well for single values, it can cause a problem when there are multiple values. If the responding node is no longer one of the closest to the key being searched for, then the values it is returning will probably be the staler ones in the system, as it will not have the latest stored values. However, the search for closer nodes will stop here, as the queried node only returned values and not a list of nodes to

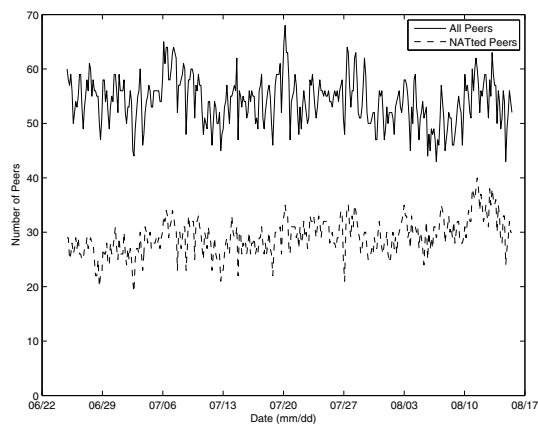


Fig. 6. The number of peers found in the system, and how many are behind a firewall or NAT.

recursively query. We could have the request return both the values and the list of nodes, but that would severely limit the size and number of the values that could be returned in a single UDP packet.

Instead, we have broken up the original `find_value` operation into two parts. The new `find_value` request always returns a list of nodes that the node believes are closest to the key, as well as a number indicating the number of values that this node has for the key. Once a querying node has finished its search for nodes and found the closest ones to the key, it can issue `get_value` requests to some nodes to actually retrieve the values they have. This allows for much more control of when and how many nodes to query for values. For example, a querying node could abort the search once it has found enough values in some nodes, or it could choose to only request values from the nodes that are closest to the key being searched for.

## VI. PERFORMANCE EVALUATION

Our `apt-p2p` implementation supporting the Debian package distribution system has been available to all Debian users since May 3rd, 2008 [11], and is also available in the latest release of Ubuntu [12]. We created a *walker* that will navigate the DHT and find all the peers currently connected to it. This allows us to analyze many aspects of our implementation in the real Internet environment.

### A. Peer Lifetimes

We first began analyzing the DHT on June 24th, 2008, and continued until we had gathered almost 2 months of data. Figure 6 shows the number of peers we have seen in the DHT during this time. The peer population is very steady, with just over 50 regular users participating in the DHT at any time. We also note that we find 100 users who connect regularly (weekly), and we have found 186 unique users in the 2 months of our analysis.

We also determined which users are behind a firewall or NAT, which is one of the main problems of implementing a peer-to-peer network. These peers will be unresponsive to

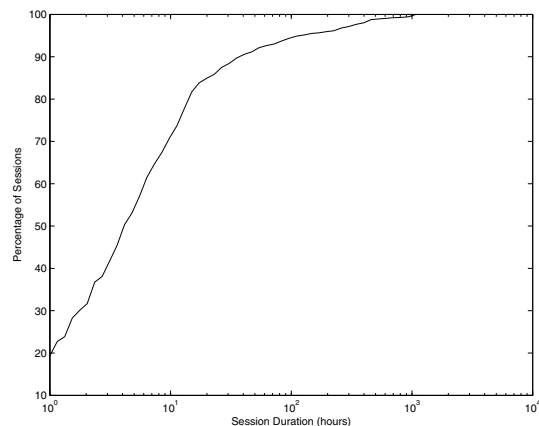


Fig. 7. The CDF of how long an average session will last.

DHT requests from peers they have not contacted recently, which will cause the peer to wait for a timeout to occur (currently 9 seconds) before moving on. They will also be unable to contribute any upload bandwidth to other peers, as all requests for packages from them will also timeout. From Figure 6, we see that approximately half of all peers suffered from this restriction. To address this problem, we added one other new RPC request that nodes can make: `join`. This request is only sent on first loading the DHT, and is usually only sent to the bootstrap nodes that are listed for the DHT. These bootstrap nodes will respond to the request with the requesting peer's IP and port, so that the peer can determine what its outside IP address is and whether port translation is being used. In the future, we hope to add functionality similar to STUN [13], so that nodes can detect whether they are NATted and take appropriate steps to circumvent it.

Figure 7 shows the cumulative distribution of how long a connection from a peer can be expected to last. Due to our software being installed as a daemon that is started by default every time their computer boots up, peers are expected to stay for a long period in the system. Indeed, we find that 50% of connections last longer than 5 hours, and 20% last longer than 10 hours. These connections are much longer than those reported by Saroiu et al. [14] for other peer-to-peer systems, which had 50% of Napster and Gnutella sessions lasting only 1 hour.

Since our DHT is based on Kademlia, which was designed based on the probability that a node will remain up another hour, we also analyzed our system for this parameter. Figure 8 shows the fraction of peers that will remain online for another hour, as a function of how long they have been online so far. Maymounkov and Mazieres found that the longer a node has been online, the higher the probability that it will stay online [7]. Our results also show this behavior. In addition, similar to the Gnutella peers, over 90% of our peers that have been online for 10 hours, will remain online for another hour. Our results also show that, for our system, over 80% of all peers will remain online another hour, compared with around 50% for Gnutella.

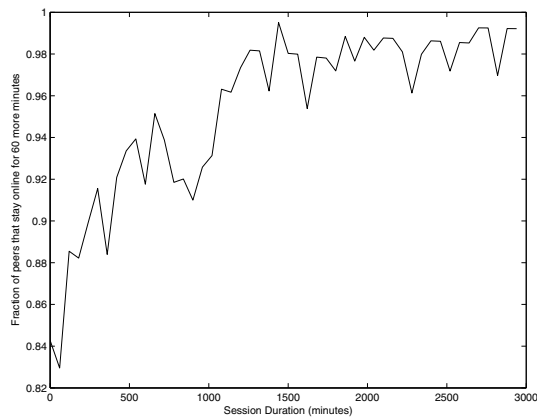


Fig. 8. The fraction of peers that, given their current duration in the system, will stay online for another hour.

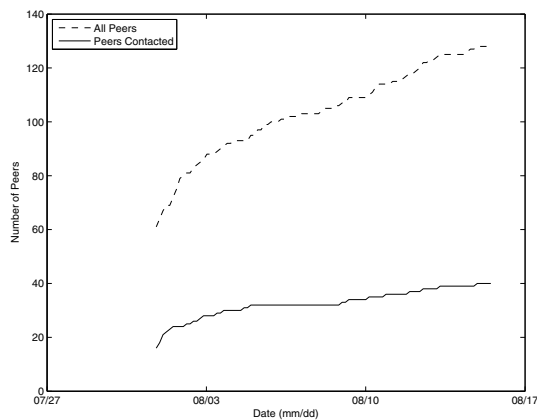


Fig. 9. The number of peers that were contacted to determine their bandwidth, and the total number of peers in the system.

### B. Peer Statistics

On July 31st we enhanced our walker to retrieve additional information from each contacted peer. The peers are configured, by default, to publish some statistics on how much they are downloading and uploading, and their measured response times for DHT queries. Our walker can extract this information if the peer is not firewalled or NATted, it has not disabled this functionality, and if it uses the same port for both its DHT (UDP) requests and download (TCP) requests (which is also the default configuration behavior).

Figure 9 shows the total number of peers we have been able to contact since starting to gather this additional information, as well as how many total peers were found. We were only able to contact 30% of all the peers that connected to the system during this time.

Figure 10 shows the amount of data the peers we were able to contact have downloaded. Peers measure their downloads from other peers and mirrors separately, so we are able to get an idea of how much savings our system is generating for the mirrors. We see that the peers are downloading approximately 20% of their package data from other peers, which is saving the mirrors from supplying that bandwidth. The actual num-

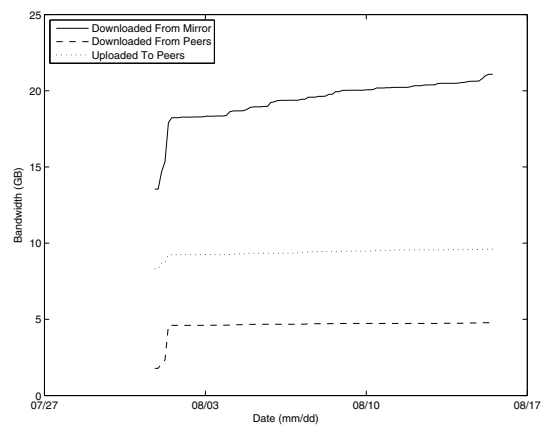


Fig. 10. The bandwidth of data (total number of bytes) that the contacted peers have downloaded and uploaded.

bers are only a lower bound, since we have only contacted 30% of the peers in the system, but we can estimate that apt-p2p has already saved the mirrors 15 GB of bandwidth, or 1 GB per day. Considering the current small number of users this savings is quite large, and is expected to grow considerably as more users participate in the P2P system.

We also collected the statistics on the measured response time peers were experiencing when sending requests to the DHT. We found that the recursive `find_value` query, which is necessary before a download can occur, is taking 17 seconds on average. This indicates that, on average, requests are experiencing almost 2 full stalls while waiting for the 9 second timeouts to occur on unresponsive peers. This time is longer than our target of 10 seconds, although it will only lead to a slight average delay in downloading of 1.7 seconds when the default 10 concurrent downloads are occurring. This increased response time is due to the number of peers that were behind firewalls or NATs, which was much higher than we anticipated. We do have plans to improve this through better informing of users of their NATted status, the use of STUN [13] to circumvent the NATs, and by better exclusion of NATted peers from the DHT (which does not prevent them from using the system).

We were also concerned that the constant DHT requests and responses, even while not downloading, would overwhelm some peers' network connections. However, we found that peers are using 200 to 300 bytes/sec of bandwidth in servicing the DHT. These numbers are small enough to not affect any other network services the peer would be running.

## VII. RELATED WORK

There have been other preliminary attempts to implement peer-to-peer distributors for software packages. apt-torrent [15] creates torrents for some of the larger packages available, but it ignores the smaller packages, which are often the most popular. DebTorrent [16] makes widespread modifications to a traditional BitTorrent client, to try and fix the drawbacks mentioned in Section II-C. However, these changes also require

some modifications to the distribution system to support it. Our system considers all the files available to users to download, and makes use of the existing infrastructure unmodified.

There are a number of works dedicated to developing a collaborative content distribution network (CDN) using peer-to-peer techniques. Freedman et al. developed Coral [17] using a distributed *sloppy* hash table to speed request times. Pierre and van Steen developed Globule [18] which uses typical DNS and HTTP redirection techniques to serve requests from a network of replica servers, which in turn draw their content from the original location (or a backup). Shah et al. [19] analyze an existing software delivery system and use the results to design a peer-to-peer content distribution network that makes use of volunteer servers to help with the load. None of these systems meets our goal of an even distribution of load amongst the users of the system. Not all users of the systems become peers, and so are not able to contribute back to the system after downloading. The volunteers that do contribute as servers are required to contribute larger amounts of bandwidth, both for uploading to others, and in downloading content they are not in need of in order to share them with other users. Our system treats all users equally, requiring all to become peers in the system, sharing the uploading load equally amongst all, but does not require any user to download files they would not otherwise need.

The most similar works to ours are by Shah et al. [19] and Shark by Annapureddy et al. [20]. Shah's system, in addition to the drawbacks mentioned above, is not focused on the interactivity of downloads, as half of all requests were required "to wait between 8 and 15 minutes." In contrast, lookups in our system take only seconds to complete, and all requests can be completed in under a minute. Shark makes use of Coral's distributed sloppy hash table to speed the lookup time, but their system is more suited to its intended use as a distributed file server. It does not make use of authoritative copies of the original files, allowing instead any users in the system to update files and propagate those changes to others. Our system is well-tailored to the application of disseminating the unchanging software packages from the authoritative sources to all users.

## VIII. CONCLUSION AND FUTURE WORK

In this paper, we have provided strong evidence that free software package distribution and update exhibit many distinct characteristics, which call for new designs other than the existing peer-to-peer systems for file sharing. To this end, we have presented `apt-p2p`, a novel peer-to-peer distributor that sits between client and server, providing efficient and transparent downloading and updating services for software packages. We have addressed the key design issues in `apt-p2p`, including DHT customization, response time reduction, and multi-value extension. `apt-p2p` has been used in conjunction with Debian-based distribution of Linux software packages and is also available in the latest release of Ubuntu. Existing real user statistics have suggested that it interacts well with clients and substantially reduces server cost.

There are many future avenues toward improving our implementation. Besides evaluating its performance in larger scales, we are particularly interested in further speeding up some of the slower recursive DHT requests. We expect to accomplish this by fine tuning the parameters of our current system, better exclusion of NATted peers from the routing tables, and through the use of STUN [13] to circumvent the NATs of the 50% of the peers that have not configured port forwarding.

One aspect missing from our model is the removal of old packages from the cache. Since our implementation is still relatively young, we have not had to deal with the problems of a growing cache of obsolete packages consuming all of a user's hard drive. We plan to implement some form of least recently used (LRU) cache removal technique, in which packages that are no longer available on the server, no longer requested by peers, or simply are the oldest in the cache, will be removed.

## REFERENCES

- [1] J. Feller and B. Fitzgerald, "A framework analysis of the open source software development paradigm," *Proceedings of the twenty first international conference on Information systems*, pp. 58–69, 2000.
- [2] (2008) Ubuntu blueprint for using torrent's to download packages. [Online]. Available: <https://blueprints.launchpad.net/ubuntu/+spec/apt-torrent>
- [3] The Advanced packaging tool, or APT (from Wikipedia). [Online]. Available: [http://en.wikipedia.org/wiki/Advanced\\_Packaging\\_Tool](http://en.wikipedia.org/wiki/Advanced_Packaging_Tool)
- [4] C. Gkantsidis, T. Karagiannis, and M. Vojnovic, "Planet scale software updates," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 423–434, 2006.
- [5] (2008) The Debian Popularity Contest website. [Online]. Available: <http://popcon.debian.org/>
- [6] B. Cohen. (2003, May) Incentives build robustness in BitTorrent. [Online]. Available: <http://bitconjurer.org/BitTorrent/bittorrentecon.pdf>
- [7] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," *Peer-To-Peer Systems: First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002*.
- [8] (2008) The apt-p2p website. [Online]. Available: <http://www.camrdale.org/apt-p2p/>
- [9] (2008) The Khashmir website. [Online]. Available: <http://khashmir.sourceforge.net/>
- [10] (2007) The PlanetLab website. [Online]. Available: <http://www.planet-lab.org/>
- [11] (2008) An overview of the apt-p2p source package in Debian. [Online]. Available: <http://packages.qa.debian.org/a/apt-p2p.html>
- [12] (2008) An overview of the apt-p2p source package in Ubuntu. [Online]. Available: <https://launchpad.net/ubuntu/+source/apt-p2p/>
- [13] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs)," RFC 3489, March 2003.
- [14] S. Saroiu, P. Gummadi, S. Gribble *et al.*, "A measurement study of peer-to-peer file sharing systems," University of Washington, Tech. Rep., 2001.
- [15] (2008) The Apt-Torrent website. [Online]. Available: <http://sianka.free.fr/>
- [16] (2008) The DebTorrent website. [Online]. Available: <http://debtorrent.alioth.debian.org/>
- [17] M. J. Freedman, E. Freudenthal, and D. Mazires, "Democratizing content publication with Coral," in *NSDI*. USENIX, 2004, pp. 239–252.
- [18] G. Pierre and M. van Steen, "Globule: a collaborative content delivery network," *IEEE Communications Magazine*, vol. 44, no. 8, pp. 127–133, 2006.
- [19] P. Shah, J.-F. Pris, J. Morgan, J. Schettino, and C. Venkatraman, "A P2P based architecture for secure software delivery using volunteer assistance," in *8th International Conference on Peer-to-Peer Computing 2008 (P2P'08)*.
- [20] S. Annapureddy, M. J. Freedman, and D. Mazires, "Shark: Scaling file servers via cooperative caching," in *NSDI*. USENIX, 2005.