RapidUpdate: Peer-Assisted Distribution of Security Content

Denis Serenyi and Brian Witten
Symantec Research Labs*

Abstract—**We describe RapidUpdate, a peer-assisted system tailored to the specific needs of distributing security content. Its unique features include being able to distribute small files while still offloading a vast majority of the distribution bandwidth, using central planning in order to maximize efficiency and meet distribution deadlines, and allowing peers to participate fully in the system even if behind a firewall or NAT device. We first describe the protocol and server scheduling algorithms, then utilize a full implementation of the RapidUpdate client and topology server to show the system meets its goals. As security software vendors face the burden of rapidly increasing content distribution load, we believe that this peer-assisted system has the potential to lower cost while increasing quality of service.**

I. Introduction

Today's security software vendors grapple with the burden of responding rapidly to viruses, worms, and other threats that emerge at an ever-increasing rate. The traditional approach is to distribute security content patches to their installed base that include virus signatures to protect against the latest threats. These patches are typically distributed either via a set of central servers or via a HTTP based content distribution network [1, 2]. As these threats evolve to spread and cause damage faster, there is increasing demand for ever more rapid dissemination of security content to minimize the window of time when machines are vulnerable. Couple that with growth in the number of machines with anti-virus software installed, and it is easy to imagine that it will become extremely costly to meet these needs via centralized distribution methods.

Peer to peer file distribution technologies, such as BitTorrent, have been employed to solve similar problems, such as distributing software updates [3]. However, several major problems stand in the way of applying this type of solution to security content. Firstly, these patches are small, often less than 200KB, and most P2P networks are not tuned to handle small files well. For instance, BitTorrent's HTTP based tracker protocol involves overhead that significantly eats into one's ability to save bandwidth, if files are sufficiently small. Secondly, security vendors need to be able to make quality of service guarantees about patch delivery. The ad-hoc nature of peer to peer distribution leads to poor quality of service under some conditions, such as when a file is first seeded or when a flash crowd has dissipated [4]. Lastly, the prevalence of firewalls and NAT devices on home and small business networks continues to grow, a trend that will accelerate as the pool of available IPv4 addresses dries up [5]. Protocols such as BitTorrent have well-documented problems with firewalls and NAT—clients suffer degraded download speeds and cannot fully participate in the system [6].

To address these limitations we have developed RapidUpdate, a peer-assisted distribution system for security content. RapidUpdate is similar to other P2P systems in that it employs a topology server used to direct peers to desired content, and specialized protocol for client-server and client-client communication. Where it is unique is that the server actively manages the file distribution process with the goal of meeting *distribution deadlines* (every client in the community should have a file by an administratively controlled time) while minimizing *vendor bandwidth* (data flowing in and out of vendor-controlled servers, specifically any HTTP file downloads for seeding the peer network, and messages exchanged with the topology server). For instance, if new security content is being generated every 30 minutes, then the administrator can supply this interval to the server, and have confidence that it will distribute each update within that interval, while offloading as much of the distribution load as possible onto the clients.

Because this system is geared towards a specific application, distributing security content, we are able to make some simplifying assumptions about the operating environment. Patch distribution systems typically involve the clients polling a server to check whether a new patch is available [7]. Security vendors typically release new content at regular intervals, whether it is hourly, daily, etc, and have their clients check for updates on that schedule as well. Therefore, we take it as a given in RapidUpdate that all clients in the community discover that they need a patch before the distribution deadline for that patch is reached, and that they do so via an out-of-band polling mechanism. Additionally, we assume that RapidUpdate is supplementing, rather than replacing, an existing distribution system, hence peer-assisted. We assume that we can instruct a client at any time to fetch the patch from this system instead of the peer network, and if we do the

* This work was supported by the United States Department of Homeland Security

client will be able to get the patch without any unreasonable delay.

RapidUpdate does not include any method for dividing files into pieces, as in BitTorrent. Given that this system is geared towards small files (200KB or less), dividing them doesn't result in any acceleration of the distribution or any savings of vendor bandwidth. Indeed, the typical piece size chosen for BitTorrent files is 256KB [8]. Additionally, RapidUpdate does not include tit-for-tat, as this requires clients to be exchanging pieces to function well. However, RapidUpdate does not assume that peers in this community are trustworthy and benevolent. An essential part of this system is an integrity check on any data received from another peer. Such a check is already common in patch distribution systems, even those based on HTTP downloads from trusted servers, and typically consists of having the client download a signature for the patch, in a file that is itself signed by the vendor [7]. Additionally, RapidUpdate includes an implementation of the EigenTrust peer reputation system to minimize delays caused by unreliable or malicious peers [9]. This paper, however, focuses exclusively on RapidUpdate's ability to meet distribution deadlines while minimizing vendor bandwidth rather than its security features, so in our experiments we assume the clients are trustworthy and reliable.

This paper is organized as follows. After a survey of related work in section 2, we describe the RapidUpdate protocol, and the algorithms employed by the RapidUpdate server in section 3. In section 4, we present the results of running RapidUpdate in a test environment. Section 5 provides the conclusion.

II. Related Work

Hybrid P2P content distribution networks are an ongoing area for research. Skevik et al. propose a hybrid where HTTP proxies assist the peer network and assist clients behind firewalls [6]. Wu et al. propose a centrally scheduled file distribution mechanism [10]. This is perhaps the most similar work to RapidUpdate in that it applies the idea of central planning to optimize the file distribution process. However, this is just a simulation study, and no attention is paid to circumventing firewalls and NAT.

There have been numerous attempts to improve on BitTorrent's performance and scalability by replacing or eliminating tit-for-tat, modifying the peer selection algorithm, and improving tracker server scalability [11, 12]. Indeed, RapidUpdate adopts similar changes for similar reasons. But unlike RapidUpdate, none of these methods are tuned to small files, include deadline driven central planning, or handle firewalls and NAT well.

Finally, RapidUpdate leverages STUN-based techniques for NAT hole punching [13, 14] that have been pioneered in IM applications such as Yahoo messenger, and VoIP applications such as Skype [15]. Skype, like RapidUpdate, includes a UDP based file transfer protocol.

III. RapidUpdate

The RapidUpdate system consists of a topology server that coordinates distribution primarily by routing clients that need a file to clients that already have it. There is also a client that interacts with the server in order to obtain content when it is available either from other peers or via HTTP from the vendor's servers.

The messaging between client and topology server is via the UDP based RapidUpdate protocol. In this protocol all communication is initiated by the client, which sends a request packet to the server. The server then responds to the client. Packet sequence numbers, coupled with a client-side retransmit timeout, are used to provide a simple reliability mechanism. This strategy has several advantages over a TCP based protocol:

1. The protocol facilitates UDP NAT hole punching so that clients can ultimately communicate directly with each other. Though TCP NAT hole punching is also feasible through STUNT or P2PNAT, these techniques are more sensitive to differences in behavior between NAT routers and therefore ultimately less reliable [16].

2. It is easier to build a highly scalable topology server, as there is no need to dedicate a socket per connected client.

3. UDP incurs lower overhead than TCP. In order to provide significant bandwidth savings even when file sizes are small, it is essential to keep the RapidUpdate protocol lightweight, so eliminating the 3-way handshake and TCP ACKs is critical.

When a client messages the server, the server initiates a *session*, and creates an in-memory client record. The record contains the client's IP address and port (known as its *tuple*), and eventually also contains a list of files the client is serving. The client is required to refresh its session by sending requests at a regular interval specified by the server called the *keepalive interval*, generally set to a couple of minutes. If the client has nothing to send it should send an empty *keepalive* packet. The session persists until the server disconnects the client by setting the disconnect flag in a response packet, or an idle timeout occurs because no packet was received.

As stated earlier RapidUpdate relies on clients discovering the presence of a new update via an out-of-band polling mechanism. When a client is ready to download and install the update, it first contacts the topology server with a *file lookup request* message, which tells the server the name of a file the client wants to obtain. The server responds with a *file lookup response* message,

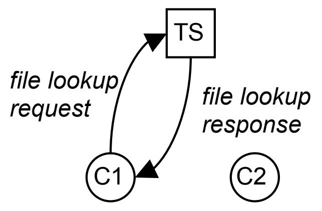

Figure 1. Client 1 obtaining a file. Response indicates it should get file from client 2

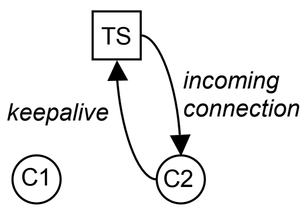

Figure 2. When client 2 next sends a keepalive, topology server instructs it to accept a connection from client 1

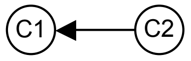

Figure 3. Client 2 sends file to client 1

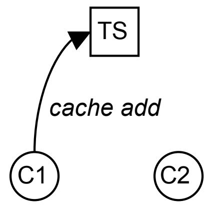

Figure 4. Client 1 sends cache add to topology server, indicating it now has the file and is available to serve it

which either provides the client with the tuple of another client to obtain that file from, tells the client to download the file via HTTP, or says to delay and retry in the next keepalive interval. When the client successfully gets a file, either via HTTP or from a client, it sends a *cache add* message to the server. After this point, the client sends keepalive messages to the server to refresh its connection timeout. The server may respond to these messages with a *incoming connection* message, instructing the client that it should expect to be used as a server, and provides the tuple of the client to expect a connection from. This process repeats until the server sets the disconnect flag in a response packet to the client, signaling that the client should cease sending packets to the server until it needs another file [figures 1-4]. For client-client communication, RapidUpdate clients use a reliable UDP file transfer protocol. If the receiver cannot contact the sender or gets poor service it will eventually time out and retry its file lookup request with the server.

This protocol works well for several reasons. Firstly, it is lightweight: typically, the client and server will exchange 8 packets total to get a file and pass it on, or about 600 bytes. Secondly, it gives the topology server flexibility in managing the client pool. If necessary, a client may be told to back off and wait before receiving the requested file, or told to get it via HTTP. A client may be asked to serve a file repeatedly at the discretion of the server. Lastly, the incoming connection message and the file lookup response message together mean that the topology server explicitly couples together client pairs before a file exchange takes place. In addition to being a good security measure (clients only accept peer connections from hosts blessed by the topology server), this a necessary step in the NAT hole punching process, as both clients must begin sending UDP packets to each other's public tuples to establish a mapping inside each other's NAT routers, thus allowing direct communication to take place [14]. Guha's recent study concludes that this technique will work with over 80% of deployed NAT routers [16].

Internally, the topology server has a linked list of available clients for each file in the system. A client is *available* if it has the file in cache and is not currently serving the file. The list is modified in response to the following events:

cache add—this results in both the client that sent the file and the client that received the file being added to the available list. If the receiver got the file via HTTP, then only the receiver is added.

file lookup request—first client in the available list is removed. This client will serve the file to the client that sent the file lookup request.

prune—the available list is used to detect spare capacity. Too much capacity is wasteful because these clients have active sessions and are sending keepalive packets regularly. In practice we've found that if the list is longer than 10% of the number of peer file transfers currently taking place, then the excess should be disconnected. The server trims the available list, and marks each client record with a flag indicating its next response packet should contain the disconnect flag.

If the available list is empty when a file lookup request arrives, which typically happens early on in the distribution (when demand for the file exceeds supply), then the server may either instruct the client to download the file via HTTP, or have it wait and retry in the next keepalive interval. Which action the server chooses to take is a function of its goal to minimize vendor bandwidth while meeting the specified deadline. To decide, it uses a simplified model of how the distribution should proceed, and then evaluates current conditions based on that model. The model we use for the distribution is a binary tree—the first client with the file passes it on to another, then the two clients pass it on to two more, and so on. The server uses the following pieces of information to calculate the size of the tree:

1. The number of total clients that need each file (# clients), necessary to calculate the tree's height. Because RapidUpdate is geared towards distributing security content to a controlled community of clients, this number can be known in advance within a small margin of error.

2. The average upstream bandwidth of each client, needed to estimate the time it takes to transfer the file from one client to the next (t_{trans}). The server tracks a running average of this number as file transfers complete. Though variance of this number can be high, if the community is large then using the average still results in a useful model.

3. The file's size, also needed to estimate t_{trans}. We assume that the file name, size, and desired distribution

deadline are delivered to the topology server at the time the file is published.

With this information, the topology server knows how much time it takes to get from one level of the tree to the next, and it knows the height of the tree, so it can estimate the total time to distribute the file (t_{distr}):

$$t_{distr} = \log_2(\text{\# clients}) * t_{trans}$$

But because t_{distr} assumes only one HTTP seed, it may exceed the deadline. By adding HTTP seeds, the server can adjust the actual completion time so that it matches up with the deadline. Looking at it another way, the server can calculate at a given point in time how many clients should have the file if we hope to meet the deadline (s_{reqd}). If fewer clients have the file, then the server can use seeds to catch up. s_{reqd} is calculated as follows:

$$t_{elapsed} = \max(0, t_{distr} - (t_{deadline} - t_{now}))$$

$$s_{reqd} = 2^{(t_{elapsed} / t_{trans})}$$

However, in most scenarios a binary tree in and of itself doesn't accurately model the entire file distribution. Rather, the first phase, where demand exceeds supply, looks like a binary tree, but once the aggregate bandwidth of the peer cloud is large enough, the rate of distribution is gated by the rate at which clients request the file. In the first phase, connected clients accumulate as they are forced to wait for their turn to get the file, and in the second phase the server's pruning algorithm acts to disconnect excess clients at the same rate that new clients request the file, keeping the number of active sessions low and constant. Figure 5 illustrates this phenomenon for an hour-long distribution where only 1 HTTP seed is used.

Given that, it actually lowers overall vendor bandwidth in some scenarios to add HTTP seeds in order to accelerate the transition to this second distribution phase. For instance, adding one additional HTTP seed would eliminate all the file lookup activity at the beginning and end of the first phase, for t_{trans} seconds. In general, adding 2^n seeds eliminates file lookups for n * t_{trans} seconds at the beginning and the end of the first phase.

To calculate approximately how many seeds are needed to minimize vendor bandwidth (s_{min}), we make some simplifying assumptions that have proven in our experiments to yield good results. Specifically, we compute each client's *lookup bw* as the size of a file lookup request and response pair, divided by the keepalive interval. We assume that on average half the clients are polling the server at any given time. Given that, s_{min} is calculated as:

$$s_{min} = (\text{lookup bw} * \text{\# clients} * t_{trans}) / \text{file size}$$

So, when the topology server processes a file lookup request with an empty available list, it compares the number of clients that have the file with s_{reqd}. If s_{reqd} is greater, then the distribution needs to be accelerated and the requesting client should download the file. Additionally, the server looks at the current number of clients that have downloaded the file via HTTP, and calculates s_{min}. If s_{min} is greater, then the requesting client should download the file, otherwise it should wait.

IV. Experimental Results

In this section, we present results from testing our implementation of RapidUpdate. We begin with a description of our implementation, followed by a description of our experimental setup, and then present our goals, methodology, and results.

A. RapidUpdate Implementation

The RapidUpdate server has been implemented in C++ and runs on Linux. We developed the RapidUpdate client as a library with a simple API (getfile, putfile), and integrated that library into a test program that instantiates up to 500 individual clients and performs a single file distribution. The clients allow the poll interval and client upstream bandwidth to be specified on the command line, and they connect and request this file at an even rate over the course of the poll interval. The server allows the distribution deadline to be specified. Finally, we developed a simple TFTP-like UDP file transfer protocol to use for client file transfers. The protocol will not work well on a WAN but was sufficient for our lab experiments.

B. Test Setup

We used 20 dual-core 2.2 GHz Intel based machines with at least 8GB of RAM each as clients, and instantiated 10 client processes for a total of 5,000 clients per machine. The server ran on similar hardware. We used RHEL 4 as our Linux distribution. The machines were connected via a Cisco Catalyst 4506 Gigabit Ethernet switch.

C. Test Goals and Methodology

Our goal was to validate our assertion that this lightweight protocol combined with the algorithms for file distribution management offloads a substantial majority of vendor bandwidth, even when distributing a small file. Secondly, we wanted to validate that RapidUpdate is able to meet or exceed the specified distribution deadline in a variety of scenarios. Thirdly, we wanted to verify that the server front-loads HTTP downloads, so as to maximize

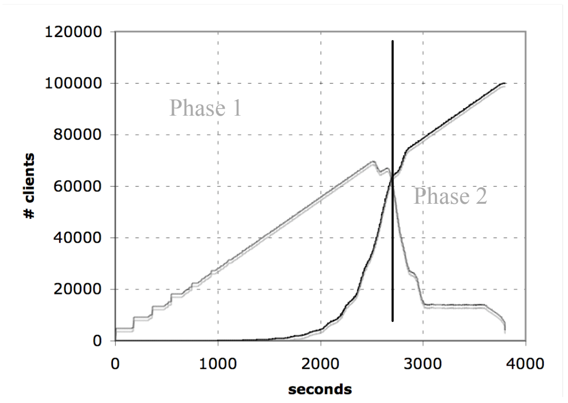

Figure 5. File distribution to 100,000 clients with 1 hour deadline, 1 HTTP seed, 3m poll interval. The black line is the number of clients that have the file, the grey line is the number of connected clients. The vertical line separates phase 1 from phase 2.

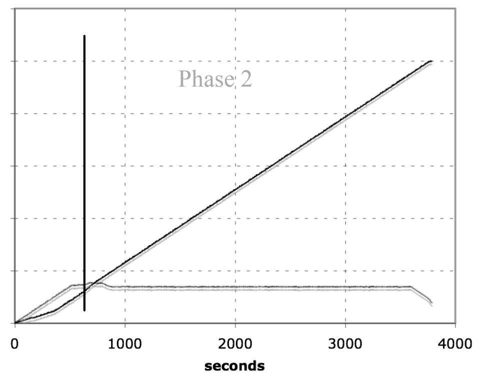

Figure 6. Same test parameters as in Figure 5, but server uses s_{min} to accelerate the transition to phase 2. Connected clients remain low throughout, significantly improving the savings in vendor bandwidth.

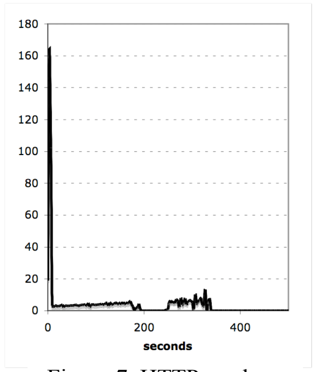

Figure 7. HTTP seeds per second (150K file, 30m deadline) as a function of time. Seeding activity is high at the beginning and quickly tails off.

their benefit and minimize their number. Finally, we wanted to verify that the server's choice of s_{min} resulted in an overall minimization of vendor bandwidth.

To measure bandwidth savings, we calculated observed vendor bandwidth after each test, and compared that to the bytes that would have been needed to distribute the file to all clients via HTTP (we did not figure in any HTTP or TCP overhead into this latter calculation). Vendor bandwidth was calculated using the number of HTTP seeds observed, and the total bytes exchanged between clients and the topology server. For reporting the latter value, the server keeps track of how many packets are received and sent, and their sizes.

We ran tests with 2 different file sizes within the range of today's security content patches (40K and 150K), and also 10K to see how RapidUpdate would perform if extremely small files were used. We used 3 different deadlines: 15, 30, and 60 minutes, which represent deadlines more aggressive than most security content distribution systems. Each client was limited to 50kb of upstream bandwidth, which is conservative assuming we are modeling clients with broadband Internet connections. In all cases we used 100,000 clients as our community size, which was at the limit of what our test network could sustain without significant congestion and packet loss. The poll interval was always set to the deadline minus 5 min. Our assumption is that administrators will want to tune their clients to request the file over a time proportional to the deadline, so scaling the poll interval in this fashion makes sense. Finally, the keepalive interval was always set to 3 min.

D. Test Results

Table 1 gives our results for these tests. In all cases RapidUpdate was able to lower vendor bandwidth substantially, even when distributing the 10K file. In fact, savings exceeded 80% except when distributing the 10K file in 15 min. Savings exceeded 90% when using the 30 min deadline, regardless of file size.

The distribution was able to beat the deadline in all of these cases because the server's initial calculation of s_{reqd} led to an accurate model of how future events would play out. Consequently, in all these tests most HTTP seeding activity occurred at the very outset of the distribution, with only minor adjustments later on, thus minimizing the total number of seeds required. As an example, figure 7 shows the rate of seeding during 30 min test with a 150K file.

Comparing figure 6 to figure 5 illustrates the improvement when the server uses s_{min} to minimize connected clients by accelerating the transition to phase 2 of the distribution and keeping the number of connected clients low. The server calculated s_{min} in this case as 2161 HTTP seeds. By adding those seeds, bandwidth savings jumped from 70.6% to 85.33%, because the number of packets exchanged with the topology server was much lower.

To ascertain whether this choice of s_{min} resulted in the greatest overall bandwidth savings, we re-ran this same test several times, forcing the server to pick a specific number

Deadline (min)	File size (KB)	% savings	Num seeds	Beat deadline by
15	150	83	16162	4 s
30	150	97.88	728	74 s
60	150	98.67	157	140 s
15	40	84.19	12434	7 s
30	40	94.46	512	98 s
60	40	95.74	552	158 s
15	10	71.31	16162	4 s
30	10	90.06	436	7 s
60	10	85.33	2161	168 s

Table 1. Test results

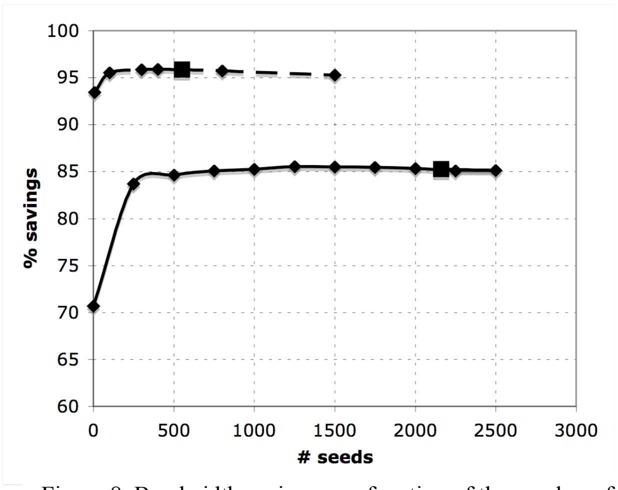

Figure 8. Bandwidth savings as a function of the number of seeds. The solid line was generated by varying the number of seeds in the 10K, 60 min test, and the dashed line was generated from the 40K, 60 min test. The square points are the number of seeds picked by s_{min}. s_{min} results in a close to optimal lowering of bandwidth.

of seeds each time. We then computed the bandwidth savings in each test, and graphed the results [figure 8]. Though the server's actual choice of number of seeds was not the absolutely optimal one, it was extremely close. We also repeated this same technique with other file sizes and the shape of the curve was the same, leading us to believe that the algorithm works well independent of the specific parameters we chose.

V. Conclusion and Future Work

We have presented RapidUpdate, the first peer-assisted patch distribution system geared towards distributing security content. Unlike other peer-assisted or P2P file distribution systems, RapidUpdate is able to offload a vast majority of the bandwidth required to distribute a file, even for small files. It additionally allows the administrator to set a distribution deadline, with the topology server automatically managing the distribution to meet the deadline. Our experiments show that the topology server's algorithms for seeding the file and managing the client pool work effectively in a controlled lab setting. Future work focuses on analyzing how well the algorithms hold up in a more chaotic WAN environment.

Bibliography

[1] http://kaspersky.com/updates.html
[2] http://symantec.com/avcenter/defs.download.html
[3] http://torrent.fedoraproject.org
[4] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, X. Zhang. Measurements, analysis, and modeling of BitTorrent-like systems. *Internet Measurement Conference*, October 2005.
[5] G. Huston. IPv4 Exhaustion Nears. *The ISP Column*, July 2007.
[6] K. Skevik, V. Goebel, T. Plagemann. Analysis of BitTorrent and its use for the design of a P2P based streaming protocol for a hybrid CDN. *Delft University of Technology Parallel and Distributed Systems Report Series*, June 2004.
[7] C. Gkantsidis, T. Karagiannis, P. Rodriguez, M. Vojnovic. Planet scale software updates. *ACM SigComm*, September 2006.
[8] B. Cohen. Incentives Build Robustness in BitTorrent. *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.
[9] S. Kamvar, M. Schlosser, H. Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. *Proceedings of the 12th international world wide web conference*, May 2003.
[10] G. Wu, T. Chiueh. How efficient is BitTorrent? *13th Annual Multimedia Computing and Networking*, January 2006.
[11] R. Sherwood, R. Braud, B. Bhattacharjee. Slurpie: A Cooperatie Bulk Data Transfer Protocol. *IEEE Infocom*, March 2004.
[12] D. Kostic, A. Rodriguez, J. Albrecht, A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. *19th ACM Symposium on Operating Systems Principles*, October 2003.
[13] J. Rosenberg, J. Weinberger, C. Huitema, R. Mahy. RFC 3489: STUN—Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs), 2003.
[14] B. Ford, P. Srisuresh, D. Kegel. Peer-to-Peer Communication Across Network Address Translators. *USENIX Annual Technical Conference*, April 2005.
[15] S. Baset, H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. *IEEE Infocom*, April 2006.
[16] S. Guha, P. Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. *Internet Measurement Conference*, October 2005.