Slurpie: A Cooperative Bulk Data Transfer Protocol

Rob Sherwood Ryan Braud Bobby Bhattacharjee
Department of Computer Science, University of Maryland, College Park, Maryland, USA
{*capveg, ryan, bobby*}@*cs.umd.edu*

Abstract—**We present Slurpie: a peer-to-peer protocol for bulk data transfer. Slurpie is specifically designed to reduce client download times for large, popular files, and to reduce load on servers that serve these files. Slurpie employs a novel adaptive downloading strategy to increase client performance, and employs a randomized backoff strategy to precisely control load on the server. We describe a full implementation of the Slurpie protocol, and present results from both controlled local-area and wide-area testbeds. Our results show that Slurpie clients *improve* performance as the size of the network increases, and the server is completely insulated from large flash crowds entering the Slurpie network.**

Keywords— **peer-to-peer communications, overlay networks, system design**

I. Introduction

Consider a situation where many Internet hosts all try to simultaneously download a large file from a central server, e.g. when a new CD image or critical patch is released for a popular operating system. As the number of clients increases beyond a critical threshold, the data rate each client receives from the server tends towards zero. When the server is so stressed, the processing and storage resources the server needs to handle client connection state is exhausted, and new clients are denied access to the server. Unfortunately, existing clients do not get adequate service either, since their data connections (using TCP) compete with each other and with new connection requests. Under severe contention on the server access link, the network regresses to congestion collapse and no client is able to make progress. Thus, it is not uncommon for extremely popular downloads to take many (tens of) hours or longer, when uncontested, the file could be downloaded in minutes. It is also not uncommon for the downloads to fail entirely, because the TCP connections either do not get created or time out due to packet losses.

In this paper, we present a protocol, named Slurpie[1], to handle this precise problem. Specifically, the goal of Slurpie is to minimize client-side wall clock time taken to download large, popular files. Our work on Slurpie is motivated by the following observation: while the resources, both bandwidth and processing, at the server are completely exhausted, the clients themselves usually have ample spare capacity. Using spare processing and bandwidth on inter-peer paths, Slurpie creates a dynamic peer-to-peer network (Figure 2) of clients who want the same file with the goal of reducing client download time and server load. Our design goals for Slurpie were the following:

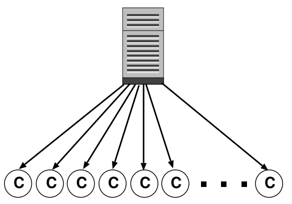

Fig. 1. Traditional data transfer: all data is transferred from the server.

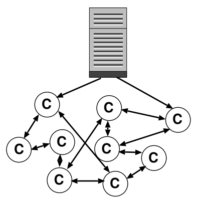

Fig. 2. Slurpie: Clients form a mesh and most data can be gotten from mesh neighbors.

Scalable: Slurpie should be scalable and robust: specifically, the protocol should be able to handle very large (10^3–10^6) simultaneous clients. Further, an explicit goal of our protocol is to maintain load at the server *independent* of the number of Slurpie clients. Thus, the entire Slurpie client set should appear as a configurable number of clients at the server, regardless of the size of the Slurpie network.
Beneficial: Clearly, the first property implies that a properly designed Slurpie protocol will reduce load at the server. However, clients will not use Slurpie unless their own download time is reduced. The second explicit design goal of Slurpie is to minimize the client download times; as we shall see, almost all clients decrease their download times by using Slurpie rather than getting the file directly from the server.
Deployable: A design goal of Slurpie is the ability to be deployed without infrastructure support. Specifically, we do not require deployment or router co-location of any new dedicated servers and it is reasonably easy for any ad-hoc groups of nodes to start their own instantiation of Slurpie. As described in the protocol description, Slurpie requires a demultiplexing host which it uses to locate other peers; we have designed the protocol such that the load —in terms of processing bandwidth, and state— on this host is minimal.

[1]Slurpie was originally designed as part of the CS 711 graduate networking course at the Univ. of Maryland.

Adaptive: Slurpie is designed to adapt to different network conditions, and tailor its download strategy to the amount of available bandwidth and processing capacity at the client.
Compatible: Lastly, we designed Slurpie such that it requires no server-side changes. In fact, a server that is serving a set of Slurpie clients cannot determine whether these clients are using Slurpie (except for the reduction in server load). Thus, Slurpie can be used with existing data transfer protocols including HTTP and FTP.

Inherent to our solution is the assumption that the server is the data transfer bottleneck, and that clients have additional resources (both processing and bandwidth) that they are willing to use to decrease their download times. Additionally, we also make the following assumptions:

- Slurpie will be used for bulk data transfer. Thus, latency and jitter are of secondary importance to overall throughput, and clients can receive and process data out of order.
- Users are *not* required to persist in the system after they finish downloading their file. Of course, system performance will increase if benevolent users choose to persist, since they can then serve parts of the the file to new users.
- An end-to-end data integrity check is available out of band. The download protocols we consider, HTTP and FTP, do not provide a cryptographically strong integrity check on transferred data. The concern due to the lack of a check is amplified when parts of the file are received from unknown nodes in the network. We assume that an application-level check is available out of band; note that this is the current norm as most popular downloads are accompanied with a MD5 checksum of the content.

A. Approach

Cooperative downloads, where the load on the server is mitigated by using other network hosts, have previously been studied and implemented in many forms. These prior efforts fall into three main categories: infrastructure-based solutions such as content-distribution networks (e.g. Akamai [1]) where server providers provision in-network hosts to alleviate load on the central server. The complementary approach is client-deployed cache hierarchies (e.g. Squid [2]) that reduce client access times (and in turn server load). There has been significant work in deploying and choosing mirror servers that replicate content. All of these approaches require fixed investment in infrastructure support and work perfectly well as long as the demand can be anticipated (and hence provisioned for). A new generation of p2p protocols (NICE [3], Narada [4], CAN-multicast [5], Scribe [6], etc.) have been developed for application-layer multicast in which streaming content is replicated and forwarded using the only resources of peers who themselves want this data. The inherent advantage of these schemes is extreme scalability. This is because, these protocols proportionately increase the amount of resources devoted to transferring data as the number of clients who want the data increase. The research focus on application-layer multicast has been on building efficient topologies that provide low end-to-end latencies. Slurpie uses this same paradigm in which peers form a dynamic structure without any extra investment in infrastructure. However, unlike prior work, our focus is on creating an efficient structure for quickly locating and disseminating bulk data. The Slurpie protocol is loosely based on the following schematic:
Suppose a popular file is available from a heavily loaded web server (called the "source server" in the rest of this paper). When a node wants to download this file, it registers with a centrally known *topology* server and retrieves a complete list of other nodes downloading the same file. The file is logically divided into fixed sized blocks, and successful completion of the download consists of downloading this set of blocks. The set of nodes downloading the same file form a per file mesh. Update messages of which nodes have which blocks are propagated through the mesh. With the update knowledge, each node can either download a given block from a peer, or from the source server.

The schematic described above is appealing, and has a number of desirable properties (e.g. reduction in server load). However, in practice, a number of problems have to be solved in order to derive a usable solution. For example, the schematic requires the topology server to maintain exact state about all peers downloading a file. Clearly, this will not scale since flash crowds of many tens of thousands can often request the same file within a very small period of time. There are many other practical problems, such as deciding on a "good" number of blocks to divide the files into, and deciding how many connections each peer should open. We also need to decide precisely how the mesh is formed, how updates are propagated, and how a peer decides to approach the server as opposed to downloading a block from the peer network. Finally, any cooperative download protocol must have a good solution for the "last block" problem, where all the nodes in the system have all but one block, and they all try to get the last block from the server! This focus of this paper is on solving precisely this set of problems, and developing a protocol that meets our stated design goals.

B. Roadmap

The rest of this paper is structured as follows: in the next section, we describe prior work, and compare Slurpie to related work. In Section III, we present specifics of the Slurpie protocol. We present experimental results in Section IV, discuss deployment issues in Section V, and conclude in Section VI.

II. Related Work

The general problem of getting popular content off of heavily loaded servers is well studied. We divide existing approaches down into categories of multicast, infrastructure-based solutions, and existing peer-to-peer efforts. We also discuss the effects of erasure encoding the data transfers.

A. Multicast

One method of reducing load at a server is to replace a number of unicast streams with one single multicast stream. This can be done either at the IP layer [7], e.g. using cyclic multicast [8], or in the application layer [3], [4], [5], [6]. The main difference between these approaches and Slurpie is that Slurpie incorporates both a *discovery* and a separate data transfer phase, i.e. in Slurpie the decision of where to get the next piece of data is made dynamically depending on network conditions and on which nodes have what data. In contrast, in all multicast-based schemes, the data source is, by default the original server, and alternate paths are used primarily for loss recovery [9], [10], [11]. Slurpie is also designed for bulk data transfer, and downloads blocks in a random order, while a number of the multicast protocols are optimized for streaming. Compared to Slurpie, most multicast protocols are much more careful about creating a topology that approximates a shortest path tree (or some some other good topological property). The Slurpie topology is essentially ad-hoc, and data transfer links are added and kept only for transferring a few blocks. We could potentially incorporate a more sophisticated topology construction algorithm in Slurpie, but Slurpie peers stay in the network for a very short period of time and our main objective in creating the topology is minimizing control overhead, and not necessarily network-level efficiency. Many (if not most) multicast protocols will not operate well if peers stayed in the network for only a few minutes, as is the norm in Slurpie. Finally, Slurpie provides complete reliability, while for the most part, reliable multicast is still has many difficult open research issues.

B. Infrastructure-based Solutions

Content distribution networks (CDNs) such as Akamai [1], [12] and web-caching hierarchies [2] are often used to alleviate load on popular servers. CDNs are deployed by the content providers (i.e. the servers), and web-caches are usually deployed by clients. A similar solution employed by some content providers is to employ a fixed number of static content mirrors (e.g. See `http://www.gnu.org/prep/ftp.html` for GNU software mirrors). Regardless of how these mirrors, caches, or CDN nodes are deployed, they are explicitly provisioned for certain load levels, and if a flash crowd exceeds this provisioned amount, then the performance of the system degrades again. In contrast, resources available to Slurpie *increase* as the client set increases, and thus, we believe Slurpie is able to handle larger client sets.

C. Peer-to-peer Bulk Transfer Protocols

Two peer-to-peer projects, CoopNet and BitTorrent, implement cooperative downloads.

C.1 CoopNet

In CoopNet [13], clients get redirect messages from the server to clients that have previously downloaded the same file. Clients are expected to remain in the system for some amount of time after they are finished downloading to serve files to future clients. The server provides multiple peers in the redirect, and an estimate of the best client is calculated. The server stores the last n (n=5–50 in simulations) clients to have requested the file, and the redirects are useful as long as one of the n clients is still serving the file. All state is stored at the server, and it is assumed that both the client and servers are CoopNet aware.

The intended application of CoopNet is downloading small HTML files, unlike Slurpie which targets bulk data transfer. There is no notion of serving a partially downloaded file, and all data transfers necessarily involve the server (in order to get the redirect list).

C.2 BitTorrent

BitTorrent [14], [15] is the work closest to Slurpie, as it targets bulk data transfer and has similar assumptions. A "tracker" service is set up to help peers downloading the same file find each other. A random mesh is formed to propagate announcements, and peers download from as many other peers as they can find. A novel feature of BitTorrent is connection choking. Peer A will stop sending blocks to peer B (this is called "choking" the connection) until peer B sends A a block, or a time out occurs. The choking encourages cooperation, as well as implicitly rate limits the data going out of a loaded peer. It is assumed that a BitTorrent client was started a priori on the web server, and that the client stays in the system indefinitely serving the file. The web server itself serves a file with a ".torrent" extension, which contains both a set of hashes for the files contents, and a URL for the tracker. From the BitTorrent documentation, it is not clear how much state the tracker keeps, but from examining the source, it appears to be $O(n)$, where n is the number of nodes downloading the file.

Compared to Slurpie, BitTorrent does not adapt to varying bandwidth conditions, or scale its number of neighbors as the group size increases. Each client appears to keep $O(n)$ state, and they periodically reconnect to the tracker to provide update information. The tracker system limits the scalability of the system to the order of thousands of nodes [15]. In Section IV, we present performance comparisons that show that Slurpie out performs BitTorrent, with respect to both average download times and also download time variance.

D. Erasure Encoding

Erasure codes have been used to efficiently transfer bulk data [16], [17]. With modest overhead, they have the benefits of resilience to packet loss and eliminate the need for stateful data transfers.

As pointed out in [16], the limitations of a stateful system, like Slurpie, typically include: lack of data distribution, per connection state, and the "last block" problem. Slurpie explicitly addresses each of these concerns via random block selection, fixed state per node, and backing off from the webserver, respectively. Finally, it is possible to incorporate erasure coding and similar encodings into Slurpie to potentially further improve performance. This is an avenue of future work.

III. Slurpie: Protocol Details

The Slurpie protocol implements the basic schematic introduced in Section I, but includes a number of refinements that are necessary for proper functioning with large client sets. At a high level, all nodes downloading the same file initially contact a topology server (Figure 3). Using information returned by the topology server, the nodes form a random mesh (Figure 4), and propagate progress updates to other nodes (Figure 5). The updates contain information about which blocks are available where, and this information is used to coordinate the actual data transfer (Figure 6). Slurpie uses an available bandwidth estimation technique, described below, that returns three states: *underutilized*, *throttle-back*, and *at-capacity*. Using this information, the protocol makes informed decisions about the number of edges to keep in the mesh, the rate at which to propagate updates, and the number of simultaneous data connections to keep open. Slurpie coordinates group downloading decisions without global information by employing a number of techniques, such as a random *back off* which controls load at the source server. It is not feasible for Slurpie clients to keep per-peer state for large download groups; we employ a mesh size estimation technique to determine the mesh size using only data stored locally. In the rest of this section, we describe different components of the Slurpie protocol, beginning with the mesh formation.

A. Mesh Formation and Update Propagation

The join procedure discussed in Section I did not scale because it assumed that the topology server kept state for the entire set of nodes downloading the same file. However, note that given a single *seed* node downloading the same file, a newly joined node can receive updates from that seed, and use the update messages to discover new peers and add new edges in the mesh. Thus, the topology server only needs to maintain information about a single node that is currently downloading a file (instead of all nodes that are downloading the file). But the question then becomes: which node id. does the topology server store, and how does it guarantee that the node is still in the system? In Slurpie, we always return the identity of the *last* node to query the topology server (for that same file). The intuition is that the node that most recently started downloading a file is the node that most likely to be still in the system. In practice, the topology server maintains and returns the last ψ nodes, where ψ is a small constant. Note that this procedure is identical to the mesh joining procedure in Narada [4].

Given a set of seed nodes, the newly joined node makes bi-directional "neighbor" links to a random subset of these nodes. Each node has a target number of neighbors (η) that it seeks to maintain. The value of η is continually updated depending on available bandwidth, and as new neighbors are discovered. The bandwidth estimation algorithm is run once a second, and if it consistently returns *underutilized*, a new neighbor, picked uniformly at random from the set of known peers, is added. In general, each node tries to maintain $\eta \geq O(\log n)$, where n is the estimated size of the total number of nodes in the mesh. Since the mesh is, at a first approximation, a random graph, the $O(\log n)$ degree implies that the mesh stays connected with high probability [18].

A.1 Update Propagation

Along each neighbor link, update messages of the form $\langle$ IP-addr, port, block-list, hopcount, node-degree $\rangle$ are passed. These form the basic information units that alert peers of new nodes joining the system, and of who has which blocks. The rate of updates passed along each link per second, σ, is subject to an AIMD flow control algorithm [19], [20] which additively increases and multiplica-

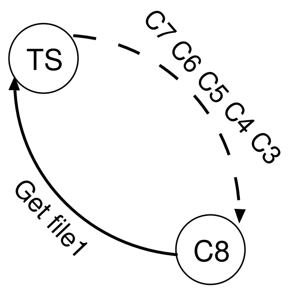

Fig. 3. Get seed nodes from topology server; topology server keeps constant per file state.

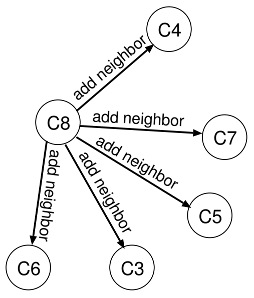

Fig. 4. Discover alive peers and form mesh; mesh degree depends on number of peers.

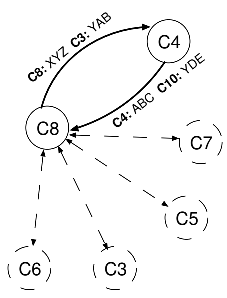

Fig. 5. Exchange updates with mesh peers; update rate controlled by bw adapatation alg.

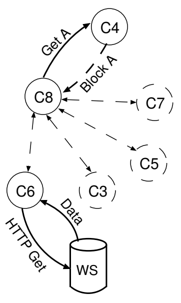

Fig. 6. Data Transfer. Server visited only if no peer has needed block.

tively decreases update rates depending on available bandwidth estimates. The intuition behind controlling the update rate in this manner is the following: when a node does not have enough peers to download from to fill its bandwidth capacity, it should increase its knowledge of the world (and thus increase the rate at which it receives updates). Correspondingly, as the node's bandwidth becomes consumed with useful data downloads, information about other peers becomes less useful.

A.1.a The Update Tree. In Slurpie updates, the block list is simply represented as a bit vector. There are certainly a number of more sophisticated data structures, e.g. Bloom filters [21] and approximate reconciliation trees [16] that we could use, for our purposes a simple bit vector has been sufficient.

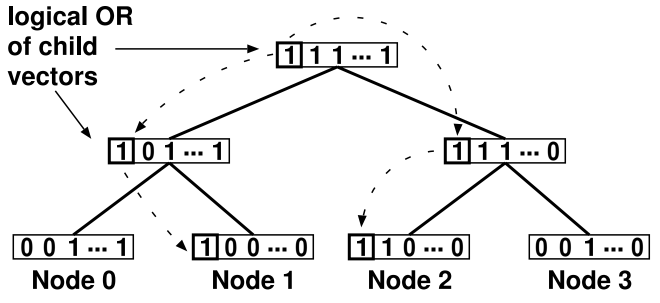

Fig. 7. Update Tree: nodes with block zero are highlighted

Each node stores information about U other nodes, where U is a constant chosen locally. The bit vectors within an update are locally stored in a data structure known as the *update tree* (see Figure 7). Bit vectors corresponding to individual nodes form the leaves of the tree, each parent is a bit vector of the logical OR of its children, and the root of the tree is the logical OR of all updates. This structure can then be used to efficiently answer queries of the form "which blocks have not been retrieved from the web server", and "which set of machines has downloaded this specific block". Only a single bit vector is stored for any peer, and newer vectors from a peer (with more bits set) replace any existing vectors from this peer. The least hop count for a given node id is also saved; this approximates the shortest path to the node, and is used in estimating the mesh size (described next).

B. Group Size Estimation

A number of the algorithms that Slurpie uses assumes that we know n, the total number of nodes downloading a given file, so it is important to be able to accurately estimate that number. Recall that U is the number of updates that any node stores. If $n \leq U$, then as time progresses and updates propagate, each node receives information about every other node in the system, and can very accurately estimate n. However, the case where $n > U$ is more interesting.

We know from random graph theory that for an r-regular graph, the mean distance d between nodes is proportional to $log_{r-1}n$. Solving this equation for n, we get $n = O((r-1)^d)$. The mesh formed by Slurpie is not exactly an r-regular graph, as nodes have different numbers of edges, and it is impossible for a single node to know the exact distance counts to all nodes in the system when $n > U$. However, using the U updates in the update tree, it is possible to estimate averages for both hop counts and degrees to gain estimates for d and r, and thus an estimate for n. Note that such an estimate becomes more accurate as n increases. In Section IV, we show that in simulations, this approximation provides reasonable estimates for n, even for relatively small values of U.

C. Downloading Decisions

In Slurpie, blocks served by peers are downloaded before blocks served by the source server. When multiple peers have the same block, we choose a peer uniformly at random. In an effort to take advantage of an open TCP window, once a connection to a peer has been established, the node downloads any blocks that it does not have from that peer.

In general, multiple downloading connections are opened in parallel, and it is a non-trivial question to decide how many connections is optimal. Here, Slurpie again makes use of the bandwidth estimation algorithm. The algorithm is queried every second, and if it returns *underutilized*, and there exist hosts that have blocks that the local node does not have, a new connection is opened.

D. Backing Off

Slurpie nodes only connect to the server if they have excess capacity, and know of no other peers that can provide them useful data blocks. Recall, however, that a design goal of the Slurpie protocol is to control the load on the source server independent of the number of peers in the Slurpie mesh. We ensure this constant load property by employing a random backoff, and in effect, system throughput *increases* as peers do *not* go to the server, even if the server is the only node that has a block they need. This is because if a large enough set of nodes opened simultaneous connections to the server for even a single block, none of the nodes would get their data, and overall system throughput would tend to zero.

Ideally, the host with the best connection to the server would be the sole machine connected to the server, and everyone else would receive their data from this host. There are, however, two problems with this method:

- The best host could download the data and then leave the system, and the entire process would have to repeat again; and
- Finding the best host is probably difficult, especially since this has to be determined quickly, dynamically, and without server support, and without probing the server (path).

Instead, we use the following scheme: Every time period τ, each eligible peer decides to go to the server with probability k/n where n is the estimate of the nodes in the system, and k is a small constant. The effect is that, on average, there will be k connections from the Slurpie mesh to the server at any time, and the number of connections to the server over time is exactly modeled by a binomial distribution with mean k. Intuitively, $k = 1$ is optimal, as it is closest to the ideal on average. However, setting $k = 1$ is too pessimistic, and results in *no* connections at the server for extended periods of time (about 30% of the time). In practice, we choose $k = 3$, which assuming $k << n$ implies there is at least one connection at the server about 90% of the time.

If we view this backoff scheme as essentially time division multiplexing, then the parameter τ becomes the length of the time slice. Logically, τ should be chosen to be long enough to guarantee some amount of progress, but short enough to ensure some amount of fairness. In this way, even a set of hosts with diverse bandwidth resources can make progress, as statistically over long downloads all hosts will eventually fetch some blocks from the server.

E. Block Size

The number of blocks a file is divided into presents a trade off between download parallelism and overhead. A small number of blocks is more efficient since it allows TCP connection overheads to be amortized, but smaller blocks reduce parallelism. As number of blocks increases, the size of the bit vector and the Slurpie control overhead increases. Instead of picking the number of blocks, we choose a fixed block size, 256KB, and let the number of blocks vary with the size of the file. We chose 256KB after conducting experiments on an unloaded system with different block sizes. A 256KB block was the smallest size at which the TCP overhead was effectively amortized ($< 1\%$). Further, the 256KB block size keeps the bit vector to a manageable size for large files (50 bytes for a 100MB file).

F. Bandwidth Estimation Technique

Slurpie requires that the bandwidth estimation algorithm only report three different states: *underutilized*, *at-capacity*, and *throttle-back*. The main design criterion of our bandwidth estimation algorithm is efficiency: Slurpie peers cannot use expensive probes [22], [23], [24] to determine precise bandwidth usage or availability. Instead, the following simplistic approach suffices: we assume that the user inputs a coarse grained bandwidth estimate of the form “Modem”, “T1/DSL”, “T3”, etc... that forms the initial maximum bandwidth estimate B_{max}. Next, we measure the sum of actual achieved throughput over all data connections over a 1 second interval, and label that B_{act}. We maintain a moving average of successive B_{act} values, calculating an *average* throughput, and the standard deviation *std* of that distribution. Using these numbers, if B_{act} drops more than one standard deviation than the average, we report throttle-back. If B_{act} is more than one standard deviation less than B_{max}, we report underutilized, else we report at-capacity. If at any time $B_{act} > B_{max}$, we set $B_{max} = B_{act}$.

IV. Experiments

In this section, we present results from our implementation of Slurpie, and compare against existing protocols.

We begin with a description of our implementation (Section IV-A), and describe our experimental setup next.

A. Slurpie Implementation

Slurpie has been implemented in multi-threaded C on the GNU/Linux system. It currently has a command line interface similar to the popular program *wget* [25], taking a URL and various options as parameters. The source code is available from the Slurpie sourceforge project[26], and should be portable to a number of platforms.

B. Experimental Setup

We experimented with Slurpie on two different networks: one on the local area network the other on the wide-area network. We used a 48-node local testbed for runs where we could precisely control the background traffic. These experiments were useful to precisely quantify Slurpie overheads and benefits, and also to compare Slurpie against BitTorrent in a predictable environment. We also deployed both Slurpie and BitTorrent on the PlanetLab wide-area testbed.

B.1 Local Testbed Setup

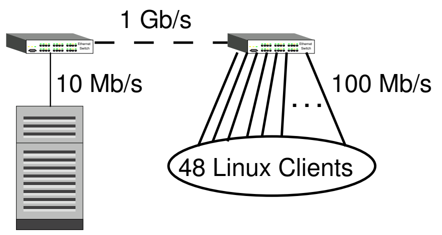

Fig. 8. Local area testbed setup. The server is connected using a 10Mbps link to force a bottleneck.

The testbed that was setup consisted of an Apache 2.0.45 web server running on an unused Linux machine with a 2.4.20 version kernel. The machine was connected to a 10Mb hub, and the the hub to a 100Mb switch, so as to force a 10Mb bottleneck at the server. The clients consisted of 48 GNU/Linux machines with 100Mb connections to a separate 100Mb switch, and the two switches were connected by a series of gigabit Ethernet links, as shown in Figure 8. Each client machine was a 650Mhz Pentium III with 768MB of RAM. In each experiment, a 100MB file was downloaded from of the web server by variable numbers of clients concurrently. The 10Mb hub is important, as by assumption, it is the server, not the client, that is the bottleneck.

B.2 PlanetLab Setup

We ran Slurpie on the PlanetLab[27] wide area network. PlanetLab consists of 55+ different sites, and 160+ different machines distributed geographically around the world. The same web server was used from the local area network tests, but with different network connectivity to the clients. From the 100MB switch connected to the web server, there is a 1Gb/s link to machines participating in Internet2, and a 95Mb/s link to machines on the general Internet. A list of machines was retrieved from the PlanetLab website, and one machine per site was chosen at random.

C. BitTorrent Setup

To compare Slurpie's performance to a comparable protocol, we downloaded the most current version of BitTorrent (version 3.2.1). To facilitate scripting, all experiments were done using Bit Torrent's "headless" mode, as opposed to GUI or Curses. BitTorrent's normal mode of operation is not to terminate after finishing downloading the file, but instead to persist indefinitely. For our experiments, we modified the BitTorrent code to terminate clients after a configurable wait after the file download is complete. In all experiments, both Slurpie and BitTorrent clients persist for the same amount of time after each experiment.

D. Results

In the results that follow, unless otherwise stated, we use the parameters listed in Table I. By default, for experiments with concurrent clients, each successive client is started 3 seconds after the previous one. (We also present results in which all clients start simultaneously). In all of the experiments, we consider the following performance metrics: total completion time and server load. The first determines client benefit from using Slurpie, and the second quantifies the benefit to the server. Finally, we present simulation results that show how our network size estimation algorithms perform.

Parameter	Description	Value
k	k/n clients go to server	3
τ	Server connection length	4 seconds
σ	Initial Update Rate	8/second
η	Initial Number of Neighbors	10
m	Mirror Time (described below)	2 seconds
U	Number of Updates Stored	100
ψ	Per File State at Topology Server	5

TABLE I
DEFAULT SLURPIE PARAMETERS

D.1 Local Testbed Results

First, we compute a baseline measure by measuring the time for a *single* client to download the 100MB file uncontested using HTTP. The baseline was measured 5 times, and the average value was used. It was assumed that all machines would have the same baseline. In the first experiment, we vary the number of concurrent clients that download the 100MB file from the server. In Figure 9, we plot the completion time for plain HTTP, BitTorrent, and Slurpie as a function of the pre-computed baseline time. For example, with 48 concurrent clients, each client, on average received only 2% of their baseline bandwidth with plain HTTP. The performance was restored to 88% with BitTorrent, and improved to 1.76 times the baseline with Slurpie. Each data point is the average measurement across active clients and then averaged across 10 runs.

As expected, these results clearly show how performance deteriorates with plain HTTP as files gain popularity. In our experiments, the BitTorrent protocol restores performance to essentially the baseline. For the vast majority of clients using Slurpie, performance *increases* as the number of peers in the network increases (recall that in these experiments, we require clients to persist only for 2 seconds after they have downloaded the entire file). Overall, this is an encouraging result indeed, and as we show later in this section, clients that join the network late are able to download the entire file at their own maximum download rate, regardless of the server capacity.

In Figure 10, we plot the cumulative distribution of the completion times of clients from the 48 concurrent node runs. Once again, each data point is an average of 10 runs. Compared to BitTorrent, Slurpie decreases average download time by 51%; more importantly, Slurpie provides more consistent performance, and the worst Slurpie client (which is the first client that joined) completes more than 5.4 times faster than the worst BitTorrent client.

To understand the steady-state dynamic of Slurpie better, we conducted a different experiment in which 245 clients joined the network, once again separated by 3 seconds each. In Figure 11, we plot the completion times of these clients. The x-axis is ordered by the order of the clients' arrival times into the system. The horizontal line is the baseline completion time (i.e. the amount of time a single client takes to download the file using plain HTTP, if there are no other clients in the system). There are several points to note: the first few clients take longer than the baseline — this is because they have to download the data mostly from the server, and pay for Slurpie overheads as well. However, once the file permeates the Slurpie mesh, the vast majority of clients get the file 2–4 times faster. There is an interesting periodic behavior evident in the completion times. This is because once the complete file is downloaded into the Slurpie network, it is distributed quickly using the mesh. However, soon clients who have the complete file leave the network (2 seconds after their download is complete), and some blocks have to be fetched from the server. This slows down completion time for a few clients who have to wait for the slow source download. However, as soon as these blocks reappear in the Slurpie network, performance increases back up until these nodes leave the network and the cycle repeats. The periodic behavior is mitigated if clients persist longer in the network.

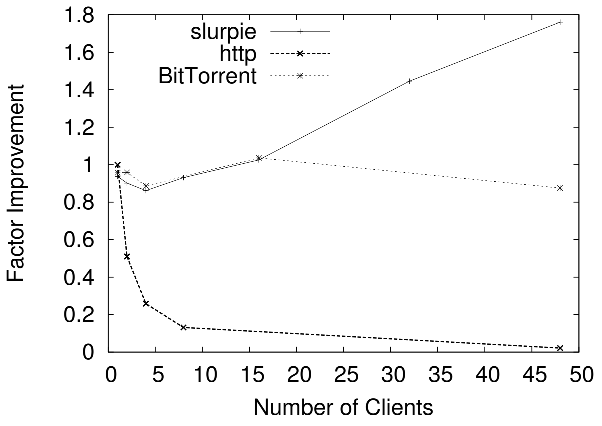

Fig. 9. Normalized completion time for varying number of clients

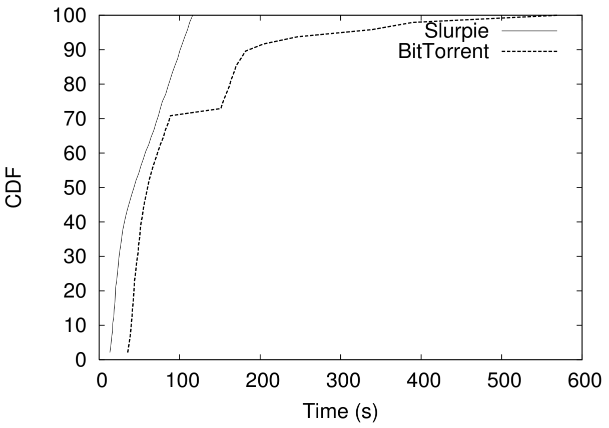

Fig. 10. CDF of completion times, 48 concurrent nodes

E. PlanetLab Results

We repeated the same experiment over the wide-area PlanetLab testbed. In Figure 12, we present the normalized completion times of varying numbers of clients using both BitTorrent and Slurpie. Once again, Slurpie outperforms BitTorrent across the client set, and our results

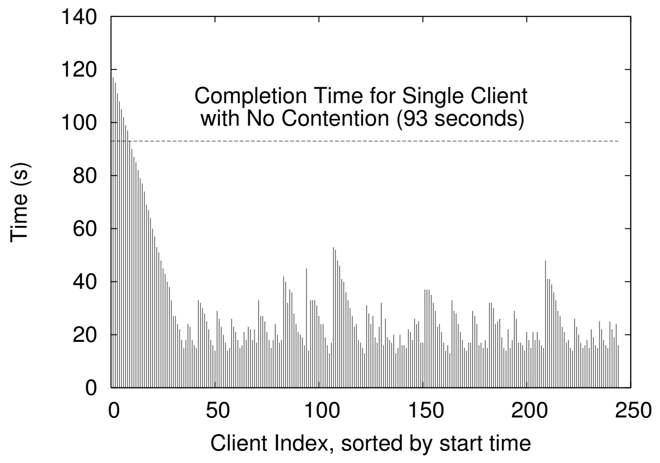

Fig. 11. Absolute completion times, 250 nodes

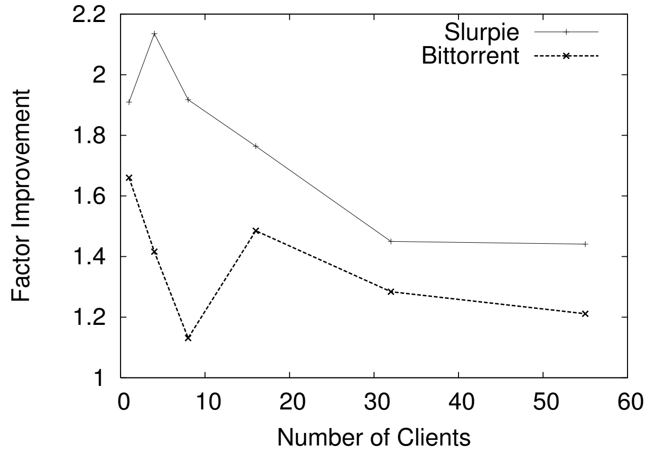

Fig. 12. Normalized completion time vs. number of clients on the PlanetLab

show that both the average and maximum time taken by Slurpie is better than BitTorrent in all runs. Note that as the number of clients increases, the relative performance with respect to the baseline reduces somewhat on the PlanetLab testbed (whereas on our local area network, the relative performance *increases*). This is because the PlanetLab hosts were being rather heavily used during the period we conducted our tests, and many of the hosts do not have much excess capacity for downloading faster from peers. Thus, as the client set increases, the number of clients with extra resources decreases as a proportion, and the average with respect to the baseline also decreases. We believe the PlanetLab hosts are uncommonly loaded compared to most Internet hosts, and in a "real" deployment, Slurpie performance would indeed increase with larger client sets.

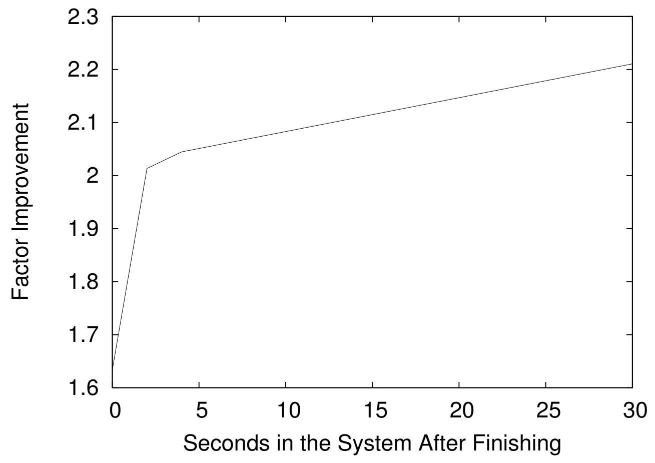

Fig. 13. Normalized completion time vs. mirror time

E.1 Mirror Time

In Slurpie, we do not require nodes to persist in the system after they finish downloading their file. It is nevertheless interesting to study the effects of benevolence, i.e. consider how completion times decrease as users stay longer after completing their download. In Figure 13, we plot completion times (again normalized against the baseline completion time), for 48 concurrent users, as users persist in the system. Interestingly, for Slurpie, almost all benefits of such mirroring is achieved if users stay in the system for only 3 extra seconds. For much larger files, we expect this number to increase, but it is clear even nominal amounts of benevolence leads to substantial benefit.

F. Coordinated Backoff

The most novel component of Slurpie is its coordinated backoff algorithm. In this section, we show how performance increases as the number of clients that go to the server is carefully controlled. In Figure 14, we plot the number of connections at the server with 48 concurrent clients with and without the backoff algorithm enabled. Without backoff, clients eventually all go the server together because some blocks are not available in the Slurpie network. The backoff algorithm carefully controls the number of clients that visit the server, and on average, the Slurpie network maintains the expected number of connections (3) to the server. Note that the number of connections drops off around 100 seconds because almost all clients complete their download by that time. As expected, the backoff algorithm controls server load. Client-side performance is also improved (Figure 15). Specifically, without backoff, the Slurpie protocol is not able to ultimately gain from the larger numbers of nodes in the network. A closer look at our data shows that without backoff, the clients all

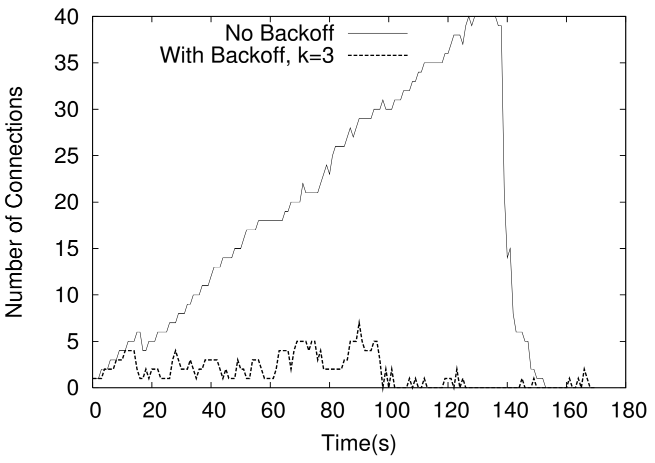

Fig. 14. Number of Connections at the server, over time

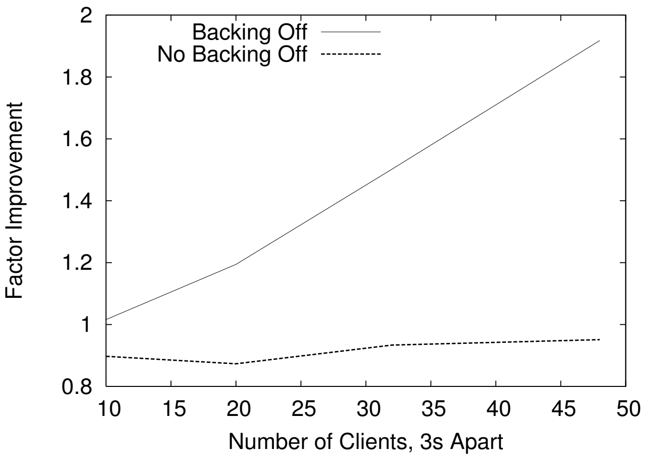

Fig. 15. Performance effects of the back off algorithm

quickly download almost all blocks, and than all visit the server for a few (sometimes just one) blocks. However, since the server is heavily loaded, all benefits from having received the other blocks quickly is negated.

F.1 Effects of Flash Crowds

In our previous experiments, we start concurrent clients 3 seconds apart deterministically. We have also experimented with random offsets between clients. However, in the worst case, all clients would start exactly at the same time. In Figure 16, we plot the number of open connections at the web server over time as the number of clients on the LAN that start *at the same time* is varied from 10–48.

Recall that a client tries to estimate the number of nodes in the mesh n, and tries to connect to the server with probability k/n, where k is set to 4. The y-axis in the plot is set to the same scale as Figure 14. Recall that in that experi-

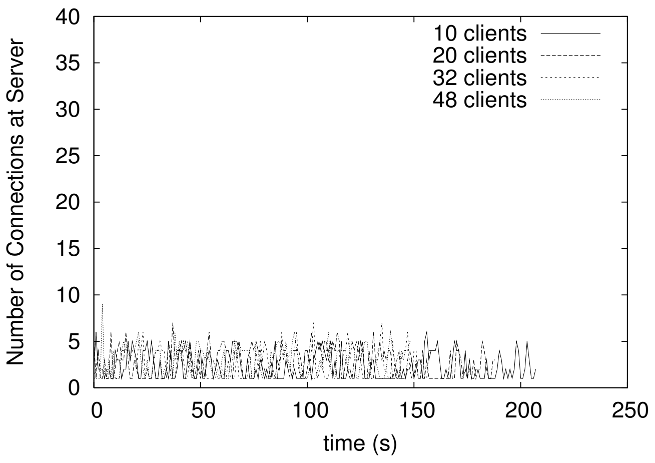

Fig. 16. Number of connections at server with different numbers of clients, all started simultaneously

ment, without backoff, even with clients started 3-seconds apart, the number of simultaneous connections increased to more than 40. In Figure 16, there are different curves for 10, 20, 32, and 48 simultaneous connections, but it is difficult to distinguish these cases. Thus, the Slurpie size estimation algorithm is effective: server load is independent of the Slurpie mesh size. We note that 48 clients arriving at exactly the same time is indicative of severe congestion (several thousand new connections per second), and Slurpie is able to easily contend with such load spikes.

G. *Group Size Estimation*

In an effort to gauge the quality of the group size estimation, we simulated the neighbor mesh algorithm with large group sizes. The simulator took three parameters, n, the number of nodes in the system, r, the target degree of each node, and U, the number of updates stored. Then, using the formula described in Section III, the simulation returned n', the average estimate of the system size. We present results in Table II for meshes with target out degree fixed at 10. The estimation error levels decrease as the state per node increases, and as the number of nodes in the system increases. This is because the estimation is derived from an asymptotic formula which provides better bounds with larger group sizes. Note that in almost all realistic scenarios, we do not expect to use the estimation with less than 1000 nodes in the system (with 1000 nodes, each client has to keep a maximum of 6MB of update state for a 100MB file). Finally, note that the backoff algorithm does not require very precise estimations of group size, e.g. estimating n with $\pm 33.3\%$ error and $k = 3$ will, on average, result in $\pm$ one extra connection to the source server.

n	20	50	100
200	17.5%	13.2%	10.9%
1000	5.9%	4.2 %	2.6%
5000	11.3%	7.5%	6.0%
10000	3.8%	0.8 %	0.4%

TABLE II
% ERROR IN GROUP SIZE ESTIMATION

V. DISCUSSION

In the results section, we have concentrated entirely on the data transfer dynamics of Slurpie. In this section, we discuss the implementation and deployment of the two other components: the topology server and security issues.

A. Topology Server

The topology server in Slurpie serves the same purpose as the rendezvous point in Narada [4] or the BSE in the NICE [3] protocol. One possible concern is the scalability of the Slurpie topology server: a scalable network does no good if clients cannot join because the server required for joining is overloaded! In practice, the topology server stores the IP address and port of the last five nodes to request a given file. This amounts to state of 30 bytes per file plus the file name, so any reasonable machine can store state for millions of files. Since the server performs no client-specific processing, the processing requirements at the server are minimal.

Of more concern is the network overhead at the topology server. Upon joining the system, every node makes a TCP connection to the topology server, tells it which file they are downloading, and then receives the IP address/port pairs of the last 5 nodes to download that same file. The entire transaction uses one packet in either direction, plus TCP overhead, so it is conceivable for a single server to handle tens of thousands of downloads per second. If the Slurpie system grows to the point where this is insufficient, the topology server functionality could be distributed. Specifically, a number of hosts that provide this service could form a DHT [28], [29], [30], and the file name could be used to look up the server responsible for the specific file. However, we do not believe the scalability of the topology server will be the limiting factor in the deployment of a system such as Slurpie.

B. Security Concerns

Using Slurpie introduces potentially new security and data integrity concerns for end hosts. In the best case, Slurpie clients will download almost all parts of files from unknown nodes on the Internet. However, we argue that this does not add significantly new security risks. A security integrity check should be performed for sensitive files, even if it is downloaded from the source server. As we mentioned in Section I, servers often publish an MD5 or similar checksum which is used to verify file integrity. Such a checksum could be used by Slurpie clients as well. It is possible for a determined adversary to attack the Slurpie network by propagating both false blocks and a corresponding *false checksum*. Note that this is a problem even in the source download case, since a determined adversary can mount any number of attacks that base IP is susceptible to, including DNS spoofing or TCP connection hijacking. The solution, of course, is to distribute a signed integrity check, where the clients can independently verify the checksum since it is signed by a trusted server. Such a solution requires an out-of-band channel by which clients get the server's public key, and once implemented, is sufficient for both plain IP and for Slurpie.

Another potential problem is a DoS attack against the Slurpie topology server. If the topology server does not function, new nodes cannot join the network. Once again, we believe this is a general problem and not specific to Slurpie, and the solutions are no different from the ones that can be employed to protect any source server.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the Slurpie protocol for scalable downloading of bulk data over the Internet. We believe the Slurpie protocol fulfills its design goals of system scalability, improved client performance, and insulation of the server from load variance in the client population. We have presented extensive experimental analysis of different components of the Slurpie protocol using a complete implementation over a local- and wide-area testbed. Specifically, our results show that client performance increases as the client population increases. This is because clients can now download parts of files from other clients without accessing highly contested server resources. Further, our results indicate that the Slurpie randomized backoff scheme is effective, and is able to precisely control server load regardless of the size or variation in client population.

There are a number of interesting open issues with Slurpie design. We believe it is possible to implement better estimates of the network size, especially if the underlying graph structure of the Slurpie mesh was studied in more detail. One problem with the current interface is it is insufficient for mass deployment, since it requires users to explicitly invoke the Slurpie protocol to download popular files. An obvious extension is to deploy a Slurpie proxy that intercepts all user download requests, and automatically routes requests for popular files to a Slurpie network. A number of the contributions of this work are independent of the data transfer path, so another avenue of research might be to implement Slurpie's data transfer using more sophisticated encoding schemes, e.g. erasure codes.

It is also worth considering schemes where (possibly with a small amount of server side assistance), clients can quickly tell whether a particular block they have downloaded is corrupt or not. It is trivial to implement such a scheme with $O(\#blocks)$ overhead, but it is not clear if an asymptotically better scheme is feasible. Lastly, our evaluation was constrained to fifty node testbeds. While this is a good beginning, evaluation on larger networks would obviously provide more compelling evidence.

REFERENCES

[1] See www.akamai.com.

[2] See www.squid-cache.org.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommreddy, "Scalable Application Layer Multicast," in *Proceedings of ACM SIGCOMM*, 2002.

[4] Y.-H. Chu, S. G. Rao, and H. Zhang, "A Case for End System Multicast," in *Proceedings of ACM SIGMETRICS*, June 2000.

[5] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Proceedings of 3rd International Workshop on Networked Group Communications*, Nov. 2001.

[6] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[7] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," in *ACM Transactions on Computer Systems*, May 1990.

[8] K. C. Almeroth, M. H. Ammar, and Z. Fei, "Scalable delivery of web pages using cyclic best-effort multicast," in *Proceedings of INFOCOM*, 1998, pp. 1214–1221.

[9] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan, "Resilient multicast using overlays," *ACM Sigmetrics*, June 2003.

[10] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "Reliable multicast framework for light-weight sessions and application level framing," in *Proceedings of SIGCOMM*, Cambridge, Massachusetts, Sept. 1995. [Online]. Available: ftp://ftp.ee.lbl.gov/papers/srm.ps.Z

[11] X. Rex Xu, A. Myers, H. Zhang, and R. Yavatkar, "Resilient multicast support for continuous-media applications," in *Proceedings of NOSSDAV*, St. Louis, Missouri, May 1997. [Online]. Available: ftp://ftp.cs.cmu.edu/user/hzhang/NOSSDAV97.ps.Z

[12] S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller, "Construction of an efficient overlay multicast infrastructure for real-time applications," in *Proc. IEEE Infocom*, June 2003.

[13] V. N. Padmanabhan and K. Sripanidkulchai, "The case for cooperative networking," in *IPTPS*, 2002.

[14] See www.bitconjurer.org/BitTorrent.

[15] B. Cohen, "Incentives build robustness in bittorrent," in *P2P Economics Workshop*, 2003.

[16] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM-02)*, ser. Computer Communication Review, J. Wroclawski, Ed., vol. 32, 4. New York: ACM Press, Aug. 19–23 2002, pp. 47–60.

[17] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM Press, 1998, pp. 56–67.

[18] B. Bollobas, *Random Graphs*. Academic Press, 1985.

[19] V. Jacobson, "Congestion Avoidance and Control," in *Proceedings, SIGCOMM '88 Workshop*, ACM SIGCOMM. ACM Press, Aug. 1988, pp. 314–329, stanford, CA.

[20] R. Jain and K. K. Ramakrishnan, "Congestion avoidance in computer networks with a connectionless network layer: Concepts,," *Proceedings of the Computer Networking Symposium; IEEE; Washington, DC*, pp. 134–143, 1988. [Online]. Available: citeseer.nj.nec.com/article/jain97congestion.html

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the Association for Computing Machinery*, vol. 13, no. 7, pp. 422–426, 1970.

[22] R. L. Carter and M. E. Crovella, "Server selection using dynamic path characterization in wide-area networks," in *Proceedings of INFOCOM*, Kobe, Japan, Apr. 1997.

[23] K. Lai and M. Baker, "Measuring link bandwidths using a deterministic model of packet delay," in *Proceedings of SIGCOMM*, 2000, pp. 283–294.

[24] A. B. Downey, "Using pathchar to estimate internet link characteristics," in *Proceedings of SIGCOMM*, 1999, pp. 222–223. [Online]. Available: citeseer.nj.nec.com/downey99using.html

[25] "See http://www.gnu.org/software/wget/wget.html."

[26] "http://slurpie.sourceforge.net."

[27] See www.planet-lab.org.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *In Proceedings of the ACM SIGCOMM 2001 Technical Conference*, 2001.

[30] A. Rowstran and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.