DRP Report: Swarming

Scalable Content For the Masses

Daniel Stutzbach
agthorr@cs.uoregon.edu
University of Oregon
December 10, 2003

I. Introduction

The capability for people to publish their thoughts, ideas, reasoning, and first-hand accounts are one of the cornerstones of a democratic society. One of the great triumphs of the Internet is the powerful ability for individuals to publish directly. They do not require the consent of an editor or publisher, just the audience. This is a boon for the unrestricted dissemination of information.

However, the publishing capabilities of the Internet are still limited. It is difficult for individuals or small organizations to publish to large audiences. The sender must provide capacity proportional to the audience size, but the common sender has only a limited budget for bandwidth. Thus, only large organizations and the wealthy can afford to publish to large audiences. This is particularly true for large multimedia files. Over the past century, audio and video content have developed an increasingly important role in our society. Affordable distribution of audio and video to large audiences would be a great win for egalitarianism and democracy.

Peer-to-peer provides this capability. By increasing scalability, it can significantly decrease cost for large servers. However, it's true benefit lies in increasing the capacity of small servers. This will allow small users to publish content to a much larger set of viewers. Consider the case of a group of musicians or a local news station. Currently, they can easily support a small number of Internet listeners. However, suppose they suddenly experience a rise in popularity. They will no longer be able to provide content. They will need to either contract with someone who can, or their content will not reach the audience. This is a loss both for the content creator and for the audience. With peer-to-peer, the system will scale and they will be able to provide the content.

II. Swarming

In this paper, we examine a particular peer-to-peer content distribution system: swarming. Before we can begin a meaningful discussion of swarming, we must first precisely define it. Swarming systems are characterized by the combination of four key features:

1. Swarming is a file transfer mechanism.
2. Swarming is peer-to-peer.
3. Swarming peers share partial content.
4. Swarming uses parallel download.

Recently, many researchers have begun to focus on peer-to-peer techniques for improving scalability. As the Internet has grown, a gradual shift in resources has occurred. In the early days, the concentration of bandwidth, storage, and processing power was near the center of the network. As consumer devices have become exponentially more powerful, this concentration has shifted to the edge. Peer-to-peer tries to leverage this movement by placing a greater burden on user systems.

Some recent file transfer protocols use the peer-to-peer paradigm. For example, in CoopNet [1] and Pseudoserving [2] the server refers clients to others who have already downloaded the file. This improves scalability by growing the system capacity proportionally to the system load. While this concept is a powerful development, swarming takes it even further, by allowing the transfer of partial content.

With existing peer-to-peer systems, a client must receive a whole file before re-serving it. This can be problematic for large files, as it may take a long time for the whole file to reach a client. Since the performance of an overloaded server goes to zero, an abrupt increase in load may cripple a server before the peer-to-peer techniques take effect. In swarming, clients begin sharing after receiving just a small piece of the whole file. This is sharing partial content. It allows clients to quickly contribute to system capacity.

Swarming also uses parallel download. This allows a client to transfer different parts of the file from different sources. In some sense, there is a separate distribution tree for each portion of a file. However, the construction of these trees is quite distinct from the multicast trees.

Parallel download has several benefits, some of which have already been demonstrated in the client-server case [3]. It makes it possible for clients to fully utilize their local network capacity; if they could be receiving the file faster, they can simply add more sources. In the event that a source leaves the network or becomes congested, the client already has several other active sources. Finally, parallel download implicitly downloads more of the file from the fastest sources.

A. Contributions

Swarming is not an idea original to my DRP. In fact, a few swarming systems have already been developed and deployed [4, 5, 6, 7]. The most well-known of these is BitTorrent [4]. While these systems do exist, few provide a technical description of their swarming protocol and, more importantly, no formal research studies have yet been published. Although swarming seems intuitive, the design of a swarming protocol is not trivial because the design space is large and there are many dynamics involved.

This paper presents the following contributions. First, it defines the design space of swarming protocols. Second, it defines a particular swarming protocol within that design space. Third, it presents the first performance evaluation of swarming delivery, using a simulation that examines a variety of workloads and swarming parameters. Fourth, it discusses details of the simulator design and implementation, critical information for anyone wishing to reproduce or build upon my work.

The results show that swarming can scalably deliver content under loads several orders of magnitude beyond what the client-server architecture can handle. Most impressively, swarming enables a server to gracefully cope with a flash crowd, with minimal effect on client performance. Finally, the results indicate that swarming spreads the load of content delivery evenly among the peers. Finally, this paper provides insight concerning the dynamic performance of the system and the impact of several key swarming parameters. This work lays the foundation for future research on swarming.

B. Contents

This report is intended to supplement the conference paper I wrote with Professor Daniel Zappala and Professor Reza Rejaie [8]. That paper includes a list of related work, the most prominent of my research results, and a discussion of their significance. This paper includes the follows details of my project:

- It describes the protocol I implemented in the simulator.
- It provides an overview of the simulator source code.
- It describes the data collection tools I created.
- It captures my results.
- It includes an extensive list of future research topics.

III. Mechanisms

Swarming relies on four primary mechanisms:

1. Peer Identification
2. Peer Selection
3. Data Division
4. Data Selection

Peer Identification is the way that peers learn about one another. Peer Selection is the operation of choosing peers. Data Division is the way a file is broken into pieces. Data Selection is how pieces are scheduled for download. The sections that follow discuss how I implemented these four basic operations.

A. Peer Identification

Peer Identification is the process of learning about additional peers. When a client starts up, we must assume that it knows of some **root server** that has the entire file. This could be a conventional web server or a node located via a peer-to-peer search mechanism. Swarming clients begin by contacting this root server, so that they can learn of one another and can begin forming peer relationships. In my system, I use a technique known as **gossiping**. Each time two peers exchange a portion of the file, they also exchange information about other hosts they know about. This process gradually floods peer information throughout the system.

An alternative approach is to learn about peers exclusively through the root server. This is the approach taken by BitTorrent; a special **tracker** program runs on the server and acts as a meeting point for peers. Its sole function is to introduce peers to one another.

In gossiping, nodes exchange tuples of peer information in between exchanges of file data. These tuples contain the following information about other peers:

- The address of another peer
- A description of which portions of the file are available from that peer
- A timestamp indicating how recent this information is

Each host maintains a cache of 64 tuples. This is a fairly small cache due to simulator memory constraints. Real systems could have much larger caches.

During each gossip, the 10 tuples with the most recent timestamps are exchanged in each direction Additionally, each peer sends a tuple about itself. This self-referential tuple always has a fresh timestamp.

To prevent excessive connection attempts to hosts that are no longer participating, peers also store and exchange information about unreachable peers. In particular, this provides a way for peers to notify the server when a host is no longer responding. Then, the server can stop advertising unreachable host to new clients.

When a host is inaccessible, a new tuple is created with a fresh timestamp and indicates that the host has no data available. The next time the peer exchanges tuples with the server, it will let the server know that the host is no longer providing data. This prevents the server from continuing to advertise the host.

B. Data Division

One of the trickiest parts of implementing swarming is keeping track of which blocks have been received. Many developers of swarming implementations have opted to split files into atomic, equal-size, independent **blocks**. When a client requests data, it requests exactly one block. If a download fails, the client must begin downloading that block from the beginning. This greatly simplifies record-keeping, at a slight cost to efficiency. For simplicity, I implemented this scheme, with a default of 32 blocks per file.

An obvious alternative to block-granularity is byte-granularity. In this case, the client keeps track of exactly which bytes of the file it has and can request arbitrary-sized chunks of data. This allows for more diverse, and more complicated, **Data Selection** algorithms, and may be a good avenue for future work on swarming.

Another alternative is to encode the data using **Forward Error Correction**, or **FEC**. This method was used by SwarmCast [6], one of the first swarming implementations available. FEC has also been shown to be useful when using parallel download from mirrored servers [9]. FEC allows the server to encode k packets as n packets where $n \gg k$. Each client then needs to receive any $\epsilon \cdot k$ packets to reconstruct the file, where $\epsilon \geq 1$. With FEC, if two peers have $\frac{\epsilon \cdot k}{2}$ packets each, it is likely they will have enough unique data to exchange and reconstruct the file. Without FEC, they almost certainly will have some duplicated data and will therefore be unable to reconstruct the file without help from additional peers. In effect, FEC increases the number of useful peers available to a client. In exchange for this advantage, using FEC incurs a cost of $k(\epsilon - 1)$ extra packets. Additionally, there is a computational penalty, that is often worse with lower ϵ.

Another interesting area of future research is determining the usefulness of FEC. This may be possible to do without actually implementing FEC. Examining the distribution of blocks among the peers would provide insight into how useful FEC could be. Also, it would be helpful to see how much time is spent looking for peers with the right blocks, instead of actually downloading.

C. Peer Selection

Once a client knows of several other peers, it must choose which ones to connect to. In this project, I used a simple heuristic that should minimize the time spent connecting to new peers: nodes add the peer with the largest number of outstanding blocks.

For example, suppose a client is downloading a file composed of 4 blocks and the it has previously downloaded just the 3rd block. It knows about Peer A who has the 1st and 4th block, and about Peer B who has the 3rd and 4th block. In this case, the client will choose Peer A since it has two blocks that the client needs, while Peer B has only one.

Another key aspect of peer selection is choosing how many peers to connect to. In my implementation, I simply made each client attempt to maintain 4 downloads at once. Whenever it has less than 4, it attempts to add another connection. There is no limit on the number of uploads a client may have going at once.

In future work, this could be optimized. A client could adjust the number of simultaneous downloads in an effort to get the greatest utilization out of its local pipe. For example, if a client has $1.5Mbps$ of download capacity, but is only getting $32kbps$ throughput, it could increase the number of peers. Unfortunately, measuring available capacity is complex.

There are many other ways to choose peers, and future work on swarming will undoubtedly include adapting server-selection techniques.

D. Data Selection

The final mechanism of swarming is Data Selection. When downloading from a peer, the client must choose which block it wants to receive. I chose to use a simple random algorithm: create a list of all blocks the peer has that are still needed, and pick one. This is the simplest way to create a good mix of blocks within the network.

Additional research should be conducted to determine how frequently block-bottlenecks are occurring. If they are common, more advanced algorithms must be developed to ensure a better distribution of blocks. For example, it may be possible to gauge determine how rare each block is and pro-actively download rare blocks. Another option is for the server-side of each connection to suggest blocks to the client. The server-side knows which blocks it has uploaded, so it could suggest infrequently uploaded blocks. Again, this may help to prevent bottlenecks from forming over certain rare blocks. Finally, FEC could be used to increase the variety of blocks from which a peer may choose.

IV. Protocol

I implemented swarming as an extension to HTTP. This allows for incremental deployment of the protocol via add-ons to existing HTTP servers and clients (i.e., web browsers). Of the deployed swarming implementations, my implementation is most similar to the Partial File Sharing Protocol (PFSP) [10], which is likewise built over HTTP. While this approach has advantages, it is problematic for NAT and firewall users. Since these users cannot receive incoming connections, they cannot share content in a straightforward way.

Another approach is to build the swarming protocol independently. This is the approach taken by BitTorrent, which uses a custom, bidirectional protocol. By

being bidirectional, BitTorrent allows NAT and firewall users to share on their outgoing connections. However, NAT and firewall users still cannot share with each other; they can only share with users directly on the Internet.

Note that I do not purport that my swarming protocol is superior to others; at the time I developed it, PFSP and BitTorrent were also just ideas on paper. This report simply documents the protocol for those interested in the details of my simulations.

It is easiest to describe my swarming protocol by first reviewing HTTP. When an HTTP client connects to an HTTP server, it begins by issuing a request similar to the one shown in Listing 1. The first line in an HTTP request contains a single-word command, the URL, and the HTTP version. In this example, I show the `GET` command, which a client issues to fetch a page. This line may be followed by headers. After the headers is a blank line, indicating to the server that the request is complete.

An example server response is illustrated in Listing 2. The first line shows the server's HTTP version and a response code. After this are a series of headers, a blank line, and the data satisfying the request, if appropriate. In this example, the server responds with a minimal set of headers. In practice, the server uses headers to provide additional information to the client, such as the age of the content. Instead of `GET`, a client may send a `HEAD` request to retrieve these headers without actually retrieving the content.

Listing 1 Client side of Client-Server Protocol

```
1   GET <URL> HTTP/1.1
2
```

Listing 2 Server side of Client-Server Protocol

```
1   HTTP/1.1 200 OK
2   Content-Length: <size>
3
4   <begin binary data>
```

My swarming protocol operates by defining additional HTTP headers. These headers are shown in Listings 3 and 4. Note that the terms "client" and "server" are from a TCP perspective; these may actually be any peers in the system. The first thing to note is that two exchanges are conducted. This is done to facilitate partial content sharing. In the first exchange, the client sends a HEAD request. This allows the client to determine precisely which parts of the file the server has. Once the client has this information, it sends a GET request for a particular part of the file.

In Listing 3, line 2 indicates that the client is not requesting any particular part of the file, and the server may suggest one. The `<URL>` on line 3 is a connect-back URL. This notifies the server where the client will store the data. The server may connect to the client there, or advertise the URL to others. Line 4 contains information about other peers, which parts of the file they have, and how up-to-date the client's information is. This is the gossiping information discussed in Section III-A. This line may be repeated several times, to provide information about several peers. Line 5 describes the parts of the file the client already has. Finally, line 6 describes the parts of the file the client is currently receiving. Lines 2 through 6 may occur in any order. Lines with empty lists are omitted entirely.

After the server responds to the `HEAD`, the same headers are transmitted in the `GET` request. The only difference is in the `Range` header. During the `GET`, the client specifies a particular range for downloading. Note that while my implementation always transfers complete blocks, the protocol itself supports byte-level granularity.

In Listing 4, lines 1 through 7 show the server's response to the `HEAD` request, while lines 8–15 show the response to the `GET`. Lines 5 and 6 contain the gossiping information. On line 3, the server uses the `Content-Range` field to suggest a block to download. Currently, this information is not actually used. Line 4 indicates the size of the range being suggested[1].

When the client submits the `GET`, the server responds with the same headers. However, lines 3 and 4 will reflect the range the client request, rather than a range the server suggests.

Note that these connections are persistent. When the server is nearly done transmitting its binary data, the client may send another request. My implementation does not actually implement pipelining. The client does not send a new request until has completely received the previous block. Because the client sends a `HEAD` first, this means there is a $2 \cdot RTT$ penalty before each block, plus the transfer time for two sets of headers in each direction.

V. Simulator Implementation

This section provides a brief overview of the source code I created. The intent is to provide a high-level description for anyone interested in looking at the code. The following subsections provide a general sense of the flow of execution, the key modules, and how the code fits together.

I used the `doxygen` [12] program to create class diagrams and HTML documentation for the code. Therefore, comments in the source frequently include

[1]This information is redundant, but is called for by the HTTP specification [11, 14.13]

Listing 3 Client side of Swarming Protocol

```
1  HEAD <URL> HTTP/1.1
2  Range: choose
3  Swarm-Have: <range>[|...]
4  Swarm-Downloading: <range>[|...]
5
6  GET <URL> HTTP/1.1
7  Swarm-My-URL: <URL>
8  Swarm-URL: <URL>; <ranges>; <timestamp>[, ...]
9  Swarm-Have: <range>[|...]
10 Swarm-Downloading: <range>[|...]
11 Range: bytes=<offset>-<offset>
12
```

Listing 4 Server side of Swarming Protocol

```
1  HTTP/1.1 206 Partial Content
2  Accept-Ranges: bytes, choose
3  Content-Range: bytes <offset>-<offset>/<size>
4  Content-Length: <size>
5  Swarm-URL: <URL>; <ranges>; <timestamp>[, ...]
6  Swarm-Have: <ranges>
7
8  HTTP/1.1 206 Partial Content
9  Accept-Ranges: bytes, choose
10 Content-Range: bytes <offset>-<offset>/<size>
11 Content-Length: <size>
12 Swarm-URL: <URL>; <ranges>; <timestamp>[, ...]
13 Swarm-Have: <ranges>
14
15 <binary data>
```

`doxygen` commands. These commands provide `doxygen` with additional semantic information for formatting the code.

A. Design Goals

Before diving into the details of the simulator implementation, it is worth discussing the initial design goals. These will assist you in seeing the motivation for my choices, as well as provide a sense of the overall structure.

The primary goal, of course, was to simulate swarming. A secondary goal was to produce a useful packet-level simulator for application-layer protocols. I chose to build this over an existing packet level simulator: *ns*, the Network Simulator. I hoped to build a general application-layer interface over *ns*-1.4's TCP. This interface could be used to implement swarming, but would also be useful for future simulations of other protocols. Version 1.4 of *ns* was selected for its scalability characteristics. Later version of *ns* are not as adept at handling large networks.

To accomplish this, I desired to approximate the Berkeley Sockets API for TCP. This is a familiar interface for network programming; it, or a close variant, is found in every major PC operating system. By using the Berkeley Sockets API, I also hoped to make it easier to create a true implementation from a simulator implementation. Ideally, the application-layer code would not include any *ns*-specific code.

Unfortunately, there were two major difficulties. First, *ns* requires heavy use of C++ classes, while the Berkeley Socket API is a C API. Second, emulating `select` requires the ability to block. Since there are multiple virtual hosts running there within simulator, this would require multiple threads of execution. However, *ns* is not a multi-threaded program.

The `select` function is one of key functions in the Berkeley Socket API. It is used to multiplex between several sockets. With `select` an application may say "Please block until one of these streams has data to be read". This is particularly important in a peer-to-peer application, where even a client must communicate with several other hosts.

B. Application Interface

My code includes four applications:

- An HTTP Server
- An HTTP Client
- A Swarming Root Server
- A Swarming Peer

This section describes the interface available to these applications. Each application is embodied in a class that derives from class `App`. The `App` class is declared in `berkeley.h`, which sets up the environment to emulate the Berkeley Sockets API. Per-node global variables are stored in arrays, indexed by the node's number. Macros are defined so the array is transparent to the application. A special global variable, `g_app_node`, stores the currently active node.

As an example, each application uses a global variable called `errno`. This "variable" is actually a macro which expands to `g_app_errno[g_app_node]`. The application need never concern itself with this expansion; it is done behind the scenes. This approach is necessary to give each node its own `errno` variable.

The header `berkeley.h` declares several other macros and functions that map the Berkeley Socket functions onto simulator functions. This includes the functions listed below:

- `read`
- `write`
- `socket`
- `connect`
- `bind`
- `listen`
- `accept`
- `close`
- `getsockopt`
- `getpeername`
- `getsockname`
- `perror`

- exit

Some of these functions are not part of the Berkeley Sockets API, strictly speaking. However, they are regular parts of the C API that need to be emulated so applications will work in the simulator environment.

Since select cannot be emulated, several other functions are declared to provide equivalent functionality. Of necessity, these use an event-driven interface. If an application needs to be notified when a socket becomes readable or writable, it must declare a AppSocket class and override the read_handler or write_handler member functions. The application can then call hook_read or hook_write to associate an instance with a particular socket. Similarly, if an application needs to be woken up after a certain time, it must derive from TimerHandler and call hook_timer. Additional functions are provided to remove hooks.

Finally, two functions are defined purely to make the simulator more efficient: skip and scribble. These functions work like read and write, but they do not actually read or write any data to memory. This is important for a file transfer application like swarming, where we want to transfer an imaginary file, but don't want to bother copying actual bytes around. These functions provide efficiency by not bothering to copy imaginary data.

C. Notable Components

In this section, I will discuss the files that I created or significantly altered to create my simulator.

C.1 Application Interface

The application interface, including the Berkeley Sockets API emulation, is provide by berkeley.cc, berkeley.h, berkeley_internal.h, socket.cc, and socket.h. These files are also responsible for keeping track of per-socket state and the per-node list of sockets.

C.2 Data Structures

The files circular.c and circular.h implement a general-purpose ring buffer.

The files freelist.c and freelist.h implement efficient allocation and deallocation routines for resources, such as socket descriptors.

The files rangelist.cc and rangelist.h are used to keep track of blocks. They include a fast block-level data structure, BlockList, and a slower, more complicated, byte-level data structure, RangeList. The byte-level granularity is built on R-Trees [13], which are implemented in rtree.c[2] and rtree.h. R-Trees are an efficient data structure for storing intervals with attributes. They are overkill for the current implementation, but may be useful for more complex Data Selection algorithms.

C.3 Protocols

I heavily modified the fulltcp.cc module that came with *ns*. In addition to significant restructuring for readability, I added support for the following features:

- Placing data inside of the TCP packets
- Closing connections
- Correct handling of dropped SYN packets

The files http.cc and http.h implement an HTTP client and server. These classes also form the base classes for the swarming implementation which are in swarm.gperf and swarm.h.

The files swarm-url.cc and swarm-url.h form support code for the swarming implementation. They provide the gossip cache functionality, which keeps track of other peers in the system. They are also responsible for choosing the next peer to connect to.

D. Refactoring

As a result of my experiences, I now feel that *ns*-1.4 is a poor candidate for application-layer simulation, even when packet-level detail is desired. It includes too many features that are not relevant to the task at hand, such as several TCP variants, a Tcl interface, different types of router queues, and instrumentation for examining link and router behavior. In part, the difficulty is due to the structure of *ns*; it is often difficult to tell which modules are orthogonal and which are intertwined. Also, *ns* does not allow for a straightforward emulation of the Berkeley Socket API.

A better approach would include a tight, finely optimized, simulator kernel with a clean TCP implementation, and a thin Berkeley Sockets API. Implementing user-space threads might be a good way to emulate select. Due to the nature of a discrete event simulator, only one thread will ever need to be awake at a time. Thus, there is no need for a complex task scheduler.

Another candidate for refactoring is the RangeList and BlockList code which keeps track of bytes and blocks. This code became a little convoluted as a result of several rewrites as my project evolved and changed.

VI. Data Management

This project generated a lot of data: roughly 3 gigabytes in over 500 heavily compressed files. Due to their scope, some discussion of their organization and post-processing is warranted.

Each data file contains a list of events in chronological order, one per line. The first item in the line describes the nature of the event, the second item is the time in seconds since the start of the simulation, and any

[2] Some of the code in rtree.c was written by James Marr to solve a programming competition problem. I found it useful and rounded out his implementation which did not include support for deletions.

remaining fields provide additional information about the event. The captured events are as follows:

- **Client-Start** is output whenever a new client is created. It includes an identifier for the client.
- **Client-Done** is output whenever a client finished downloading the file. It includes an identifier for the client and the packet loss rate.
- **Connect-Begin** indicates that one peer is trying to contact another. The two peers are identified.
- **Connect-Done** indicates that a Connect-Begin succeeded. The two peers are identified.
- **Close** indicates that a client peer is closing its connection to a server. The two peers are identified.
- **Timeout-Client** indicates that a client has not received any data from the server and is terminating the connection. The two peers are identified.
- **Timeout-Server** indicates that a server has not received any data from a client and is terminating the connection. The two peers are identified.
- **Block-Begin** indicates that one peer has begun downloading a block from another. The peers are identified, and the block is specified.
- **Block-Done** indicates that one peer has finished downloading a block from another. The peers are identified, and the block is specified.
- **Drop-In** indicates that a packet was lost on a peer's incoming link. The peer is identified.

Prior to my DRP, I had written a Python module called "psim" for farming simulation jobs out to a set of computers. I used and enhanced this module to manage my simulations. The module is fairly simple and is not too hard to adapt to other tasks. It assumes each machine has a similar directory structure, with the simulator installed in the same place. The psim module is run on one machine that accumulates the result. As input, it takes a list of simulations to run and a list of machines. It connects to the machines via ssh[3], and runs one job at a time per CPU. The simulator must print results on standard output so that psim can receive them and store them.

The psim module has room for improvement. Currently, it is not possible to add or remove jobs from the queue without restarting psim. Since restarting psim also restarts any running simulations, this is wasteful when there are long-running simulations. Ideally, psim would use a client-server architecture. The server would be a long-running process which executes jobs, while the client would send the server commands to manipulate the job queue. This would also allow a multi-user psim to be developed, so that users can prioritize the use of resources.

The results of each experiment are stored in a directory called "results", in a file named after the simulation parameters. The random number seed is included as one of the parameters, so any result file can be exactly reproduced by running the simulation again with the same parameters. The data is compressed before storage, to curb disk usage. By using ordinary files, the results may conveniently be examined with conventional command line tools.

[3]Cryptographic ssh keys must be setup in advance to allow login without a password.

I processed the results using Python scripts, divided into three stages. The first stage computes a particular measurement from a results file. I wrote scripts to compute the following measurements:

- Blocks served by each peer
- Download time for each peer
- Percentage of transfer time spent on the Last Block Problem
- Packet loss for each peer
- Number of active peers over time
- Unique peers from which a download was attempted, for each peer
- Time-averaged number of parallel downloads for each peer
- Number of unique peers downloaded from, for each peer
- Number of unique peers uploaded to, for each peer
- The mean over all peers, of any of the above measurements

The second stage organizes the measurements into a graph. It supports four types of graphs:

- Histogram
- Two-Dimensional
- Three-Dimensional
- Cumulative Distribution Function (CDF)

These generators are extremely flexible. As input, they take a pointer to one or more measurement functions. Additionally, they take a list of simulation parameters which are used as inputs to the measurement functions. In addition to the fixed values, these parameters may be x or functions in terms of x. Stage two will look for all sets of simulations results that match the fixed parameters, compute their measurements, and plot them with x on the x-axis and the results on the y-axis. This provides a powerful mechanism for creating a variety of graphs. Naturally, the histogram and CDF graphs cannot use x, while the three-dimensional plot uses both x and y.

The first stage is quite slow. Therefore, the output of the first stage is cached in a Python "pickle" file. Pickled data is a way of storing Python objects as a stream of bytes. It is similar to Java's "serialization" feature. By using the cache, the scripts can avoid computing the same measurement for the same data. This code could be further optimized by including more fine-grained caching. Currently, each measurement needs to parse the entire results file. Since each results file is quite large, it would be much faster to scan each file

only once.

Finally, another Python script generates the actual figures. It calls the second stage, sets up any relevant Gnuplot parameters, and runs Gnuplot. Each figure is generated by a single function written in Python. By default the script runs all of these functions. Alternately, it will run only the functions named via command line parameters. This provides a simple mechanism for making slight changes to individual figures. The script uses the python-gnuplot package to call Gnuplot and generates figures using the eepic format.

VII. Results

The presentation of my results is divided into three main sections:

- Methodology, which describes how I acquired my results
- Performance, which describes broadly how swarming performs
- Depth, which explores how swarming behaves in detail

A. Methodology

To study swarming, I chose simple scenarios that would allow me to isolate and examine particular characteristics. The number of possible scenarios is enormous, therefore the best approach is to alter one variable at a time and observe the response.

I performed my simulations using a single-router Internet model. In this model, the Internet cloud is approximated by a single router. Each client or server is connected directly to this router. This is a common model for application-layer, end-to-end protocols, since the interesting bottlenecks are at the edge of the network.

Initially, I used homogeneous peers modeled after cable modem users. Each user has $1536kbps$ of download capacity and $128kbps$ of upload capacity. I opted to model the root server as a high-end DSL user, with $1Mbps$ in each direction. To focus on transmission delay, I set the latency to just $1ms$. As a workload, I used a file size of 1 Megabyte. These characteristics are summarized in Table I.

TABLE I Basic Swarming Scenario

Parameter	Value
File Size	1 Megabyte
Server capacity	$1Mbps$
Client capacity (down/up)	$1536kbps$/ $128kbps$

Pessimisticly, each client exits the system once it has downloaded the file. It closes any open connections and refuses new requests for downloads. This is a conservative approach. In the real world, many clients remain up after completing their download, adding more capacity to the system.

I generated workloads by having new clients enter the system at a give rate. This rate is an average; the clients actually arrive using an exponential distribution. This distribution is used to prevent synchronization and to more closely approximate the real world. A higher rate represents a higher load. Another component of load is the file size.

I run each simulation until 6000 download completions have occurred. I treat the time before the 500th completion as a warm-up, allowing the system to reach steady-state. I found this value by examining graphs of worst-case scenarios to see where steady-state began, and rounding off conservatively.

I ran each simulation with 3 different random number seeds. For my results, I compute the mean for each run, then take the mean of the means. This yields an accurate measurement of the mean. I compute the 95% confidence interval for each mean. The intervals are all quite small, and will get no further mention. The measurements are computed by the scripts described in Section VI.

The primary performance metric is the download duration. This is the time delta between when a client enters the system and when the client acquires the whole file.

B. Performance

Except where otherwise stated, all simulations use the configuration given in Table I. The parameters of the swarming mechanisms from Section III are summarized in Table II.

TABLE II Swarming Parameters

Parameters	Value
Concurrent downloads	4
Size of gossip cache	64
Tuples in gossip message	10
Block Size	$32KB$

B.1 Scalability

The most interesting results for any file transfer protocol are tests of scalability. Figure 1 shows the time required to download a file versus the client arrival rate. For comparison, the figure also shows the results for a traditional client-server transfer. The download time for swarming increases linearly with the rate, while the download time for client-server increases super-exponentially! In fact, client-server exhibits a vertical asymptote at around 7 clients per minute. Beyond that point, clients arrive faster than the server can transfer the file. The queue of outstanding clients increases in-

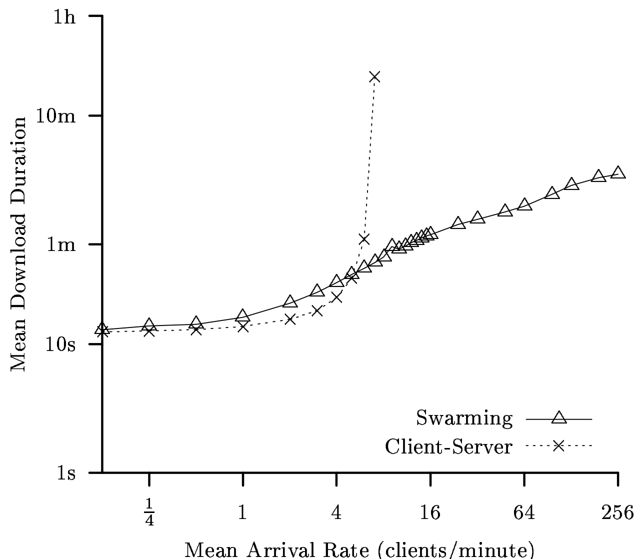

Fig. 1
IMPACT OF ARRIVAL RATE ON PERFORMANCE

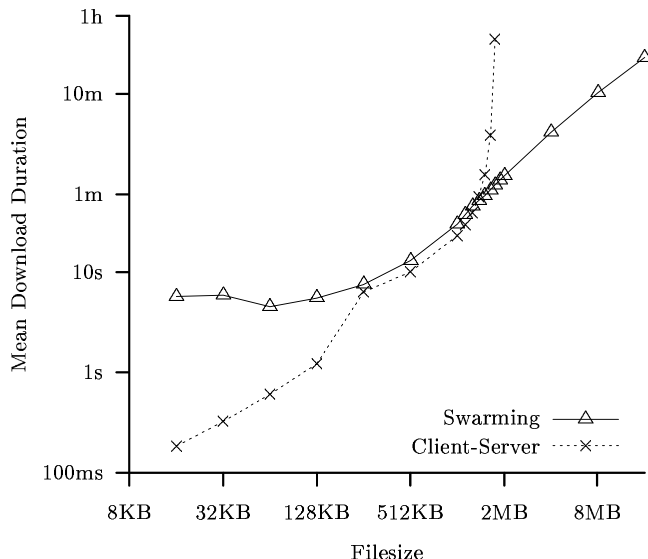

Fig. 2
IMPACT OF FILESIZE ON PERFORMANCE

definitely and the system never reaches steady-state.[4] Naturally, the point at which the client-server protocol is unable to respond will depend on server capacity, file size, and arrival rate.

Although a linear increase is shown in Figure 1, swarming must also have some asymptote. Inevitably, at some point the load will be large enough to prevent the root server from providing referrals. A back-of-the-envelope calculation suggests this will not occur for at least an order of magnitude further increase in arrival rate.[5] Unfortunately, memory limitations prevent simulation of arrival rates higher than those shown.

This asymptote exists for any scheme that relies on contacting a known, central point to initiate a download. At extremely high loads, swarming can incorporate a decentralized method for locating peers, such as the Gnutella search mechanism or PROOFS [14]. The decentralized search can locate peers who have already downloaded the whole file. As far as swarming is concerned, there is no difference between a peer with the whole file and the true root server.

These results show that swarming scales significantly better than client-server. Serving 192 clients per minute means serving the one megabyte file to more than a quarter million people per day. This is an impressive feat for a $1Mbps$ access link. To serve an equivalent load using a client-server protocol would require, at a bare minimum, $28Mbps$. This would cost thousands, perhaps tens of thousands, of dollars per month![6]

My simple implementation of swarming does impose a slight performance penalty under light load. When there are not many peers to share with, the client ends up getting most blocks from the root server, but with the added overhead of gossip messages. This seems a small price to pay for such a significant increase in capacity during high loads. Moreover, it is likely fine-tuning will eliminate the problem. For example, a server could dynamically initiate swarming only when needed. Also, pipelining could be implemented.

Another component of load is filesize. Figure 2 shows the effects of filesize on performance, using a constant arrival rate of 4 clients per minute. It has the same basic shape as Figure 1. Swarming experiences linear growth, while client-server has a vertical asymptote near a filesize of 2 Megabytes.[7]

The slight overhead of gossiping is more pronounced in Figure 2. In these simulations, I use a fixed *number* of blocks, 32, regardless of filesize. Thus, the gossip overhead is constant regardless of the filesize. Thus, as the file size decreases, the gossip message becomes

[4] In my simulations, clients never give up. They continue to retry the download until it succeeds.

[5] Assuming a single 1500-byte packet is used to transmit the referral information, the $1Mbps$ server can transmit $2^{20}/(1500 \cdot 8)$ of these per second, or 5242 per minute.

[6] http://www.bandwidthsavings.com/servicesdetail.cfm

[7] The asymptotes for client-server can easily be calculated. The arrival rate times the filesize must be less than the server's capacity. For example, with a $1Mbps$ server and 1 one Megabyte file, the asymptote should be at 7.5 client arrivals per minute. With a $1Mbps$ server and an arrival rate of 4 clients per minute, the asymptote should be at a filesize of $1.9MB$. This matches the empirical evidence I present in Figures 1 and 2.

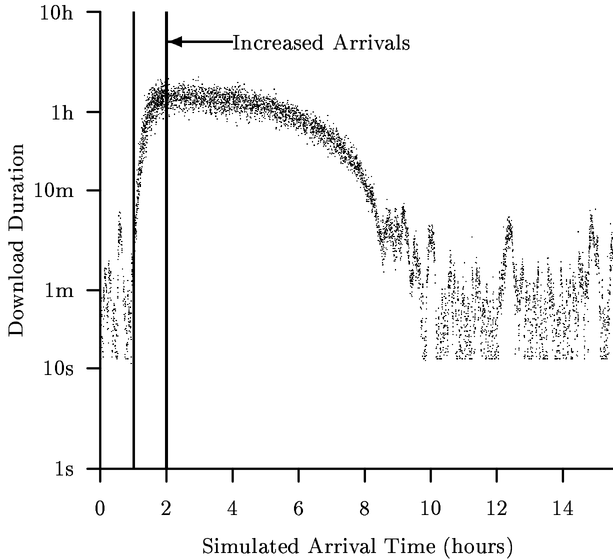

Fig. 3
CLIENT-SERVER'S REACTION TO A FLASH CROWD

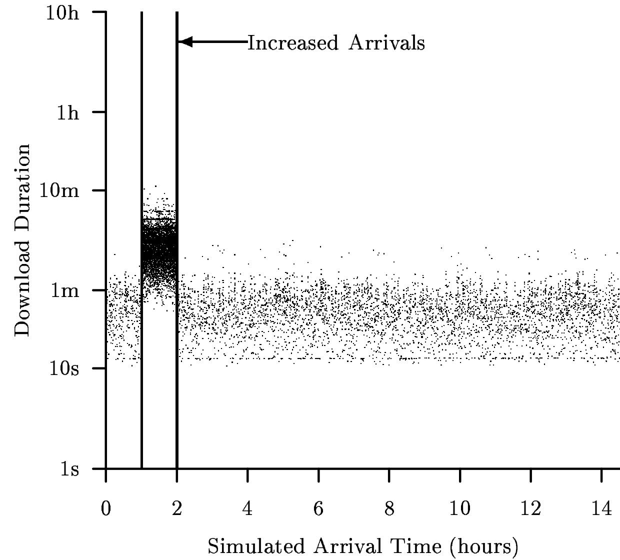

Fig. 4
SWARMING'S REACTION TO A FLASH CROWD

relatively large. This is why the mean download time in Figure 2 never goes below 4 seconds. Despite this overhead, swarming will eventually outperform client-server for smaller files as the arrival rate increases.[8]

B.2 Flash Crowd

I have shown that swarming scales excellently in steady-state. However, equally important is how quickly swarming reaches steady-state when abruptly presented with a large load. This is an important concern for a file transfer scheme, as abrupt surges in requests, called **flash crowds**, are a major concern. I simulate the effect of a flash crowd by abruptly increasing the arrival rate for a fixed period of time. The simulation begin with one hour of clients arriving at 6 per minute. The load increase for one hour, then returns to 6 per minute for the rest of the simulation.

Figure 3 shows the results for the client-server model, with a logarithmic y-axis. Each data point represents a single client. The x-axis shows the time the client arrived, while the y-axis shows the time it took the client to download the file.

Under the increased load, the crowd swells and download time grows due to an inability to service the requests. The server does not recover until long after the arrival rate decreases.

Figure 4 shows the results for swarming. It enables a web server to smoothly handle larger flash crowds that would otherwise bring content delivery to a crawl. It maintains reasonable response times as the crowd arrives and dissipates the crowd quickly.

However, this figure is slightly misleading. For client-server, the arrival rate doubled to 12 clients per minute. For swarming, the arrival rate increased an additional order of magnitude to 120 clients per minute. There are actually twice as many points on the swarming graph.

C. Depth

Now that I have examined swarming's scalability, it is time to explore the dynamics of how it distributes load, how peers interact, and how swarming's behavior is affected by certain parameters.

C.1 High Load

To examine how swarming distributes load, I examine in detail one of the simulation runs with 192 client arrivals per minute. Under this heavy load, packet loss at the root server is severe. However, it is still better than client-server, which cannot handle the load at all! Impressively, swarming still manages to get the file delivered to clients. The congestion at the root server might be relieved by adapting swarming to limit the number of concurrent uploads a server will allow. For example, it would make sense for the root server to only send data blocks to 7 clients per minute. Any additional clients would need to get blocks from peers. While this is trivial to compute when the server's capacity is known, it becomes difficult in a dynamic environment. Choosing

[8] Of course, the files need to be at least as big as a single gossip message. Swarming is of no help for tiny files.

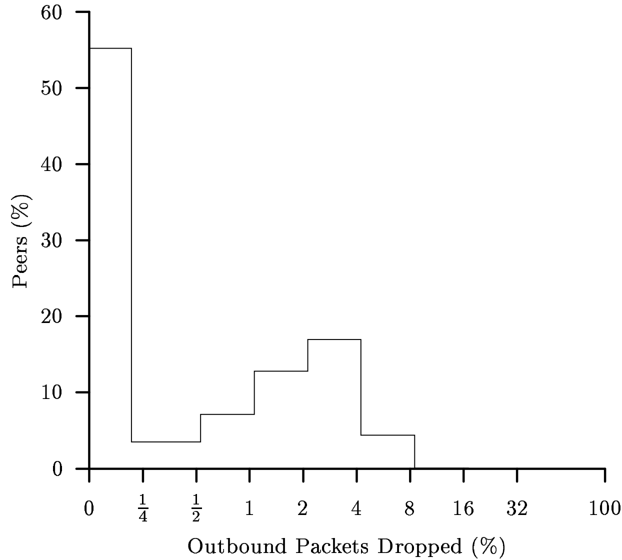

Fig. 5
Histogram of packet loss rates at 192 arrivals per minute

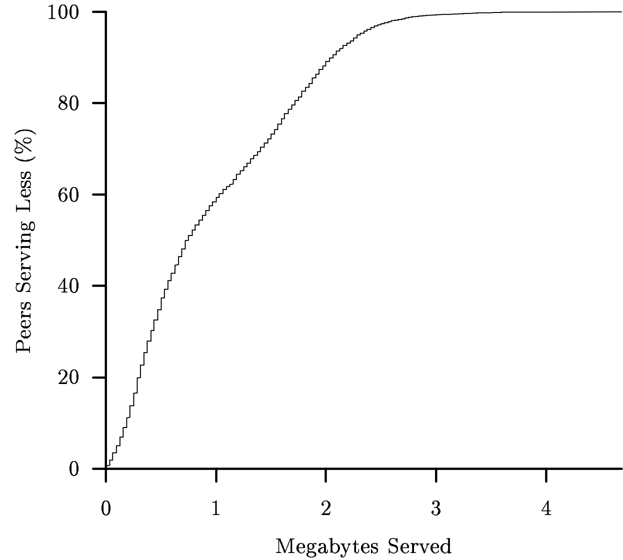

Fig. 6
CDF of megabytes served at 192 arrivals per minute

the right number of clients to serve is an area for future research.

Unlike the server, peers experience little packet loss, even at high load. This is illustrated by the histogram of peer packet loss rates shown in Figure 5, using a logarithmic scale.[9]

The low packet loss rates at the peers is due to the burden of content delivery being spread evenly among the peers. In this same high load scenario, roughly 60% of the clients serve less than one megabyte. Nearly all of the clients upload less than two megabytes. Re-serving the file once or twice is fair, so this behavior is quite good. This result is shown in Figure 6, which plots the cumulative distribution function (CDF) of megabytes served. Even if a peer has served a whole megabyte, it may not have served the whole file. It may have simply served the same block many times. This is one of the strengths of swarming; even a peer with a small portion of the file can be quite helpful.

The time for a client to complete its download is less evenly distributed. For the high load scenario, the download times are spread mostly between 1 minute and 5 minutes. However, more notably, a disproportionate number of download times are close to exact multiples of 1 minute. This is shown as a histogram in Figure 7. This behavior did not manifest at lower loads such as 16 clients per minute. It is important to note that the download times are measured individually from the start of each client; thus, this pattern does not indicate synchronization of flows within the network. After some investigation, I was able to confirm that the uneven distribution is caused by the use of a 60 second timeout to detect dead connections. Undoubtedly, these timeouts are occurring due to the severe congestion at the server. This suggests that alleviating the server congestion will also result in significant improvements for the peers.

Figure 8 shows the number of active peers over the course of the simulation. It shows the peers accumulating near the beginning, until the peers provide enough capacity to meet their own demand. Then the system enters steady-state.

C.2 Peer Dynamics

To better understand swarming's behavior, I now examine several peer-related metrics under various loads. Figure 9 plots the mean values of four per-peer metrics,:

Peers Attempted The number of unique peers contacted

Peers Downloaded From The number of unique peers from which at least one whole block was received

Peers Uploaded To The number of unique peers to which at least one whole block was sent

Concurrent Downloads The time-averaged number of parallel downloads

[9] Only outbound packet loss is shown in the figure; negligible inbound packet losses occurred. This is not surprising given the highly asymmetric capacities of the peers.

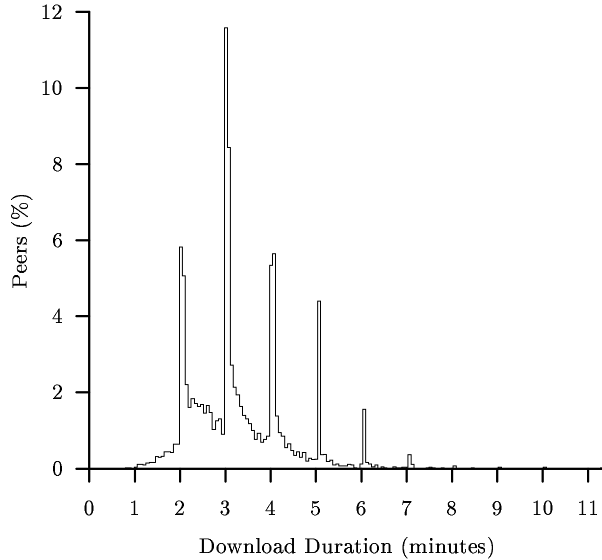

Fig. 7
Histogram of download times at 192 arrivals per minute

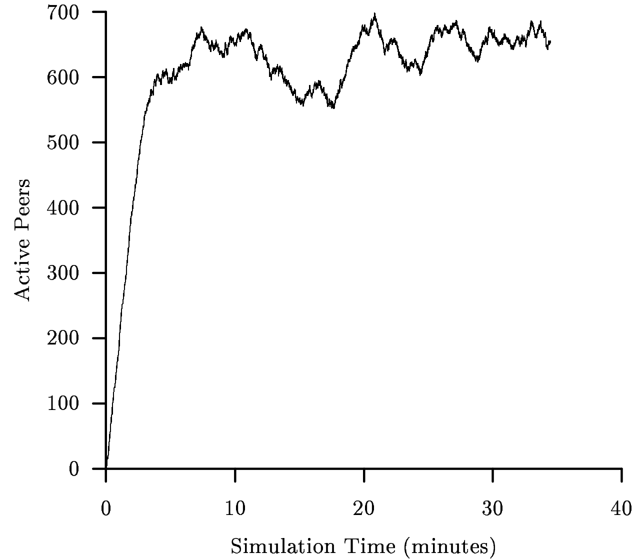

Fig. 8
Active peers over time at 192 arrivals per minute

When examining this figure, please note the logarithmic scale on the x-axis: even where the lines appear steep, the increase is quite gradual. All four metrics exhibit a rapid growth between 1 and 16 arrivals per minute, then slow. Before this interval, there is typically only one active peer at a time. However, the number of peers a client attempts to contact increases above 64 arrivals per minute. There are more active peers in the system, but each peer is downloading at a slower rate. This means that a client is contacting more peers to find the blocks it needs.

Once the arrival rate reaches 8 clients per minute, the pool of active clients is large enough that the average number of concurrent downloads for a client is close to the maximum of 4. This metric is time-average, so for example if a client spends half the simulation downloading from 3 peers and half it downloading from 4, then the time-average for the peer is 3.5.

Figure 10 plots the time-averaged count of active clients versus the arrival rate. Active clients are those that are trying to download the file, but have not yet finished doing so. Since we are looking at the steady-state behavior, this tells us how many peers must be in the system to provide the necessary capacity for that arrival rate.

C.3 Concurrent Downloads

One interesting question for swarming is whether clients are able to improve their performance by increasing the number of concurrent downloads. This is explored in Figure 12 by plotting the mean download time as a function of the maximum concurrent downloads. The swarming scenario is otherwise the default given in Tables I and II.

Under high load, Figure 12 shows a noticeable performance improvement for when increasing the concurrent downloads from 2 to 4. Just as importantly, there is no adverse impact under lower loads. Note that the increase in download time due to the change in arrival rate is consistent with Figure 1.

To explore this issue in more depth, Figure 12 plots peer dynamics versus concurrent downloads, using an arrival rate of 8 per minute. Clients are able to download from a maximum of about 8 peers at a time, even when the limit is raised to 32. Clients do in fact download from a greater number of peers as the concurrency limit increased. However, as Figure10 shows, there are only about 20 active peers at a time. Thus, clients are unable to find enough active peers with their desired blocks. It would be interesting to conduct additional simulations, varying the maximum concurrency under heavier load.

C.4 Block Size

A key parameter for swarming is block size. To fully explore the effect of block size on swarming performance I conducted a series of simulations with varying block, using an arrival rate of 16 clients per minute. Other than the block size, the rest of the scenario is again the same as shown in Tables I and II. Figure 13 shows the

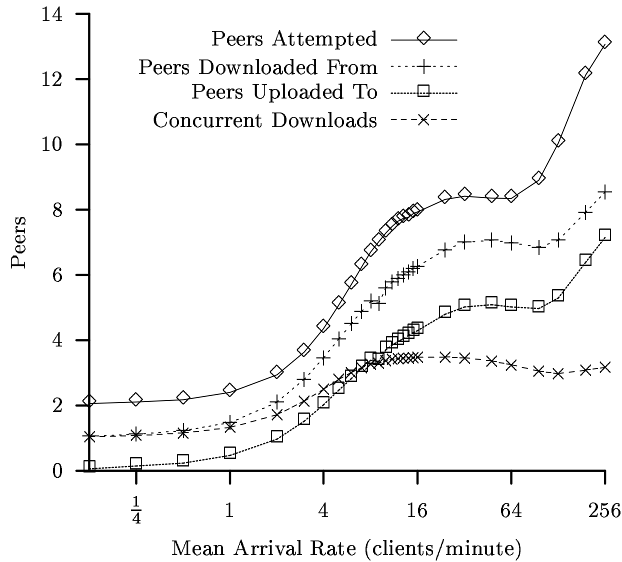

Fig. 9
Peer dynamics versus arrival rate

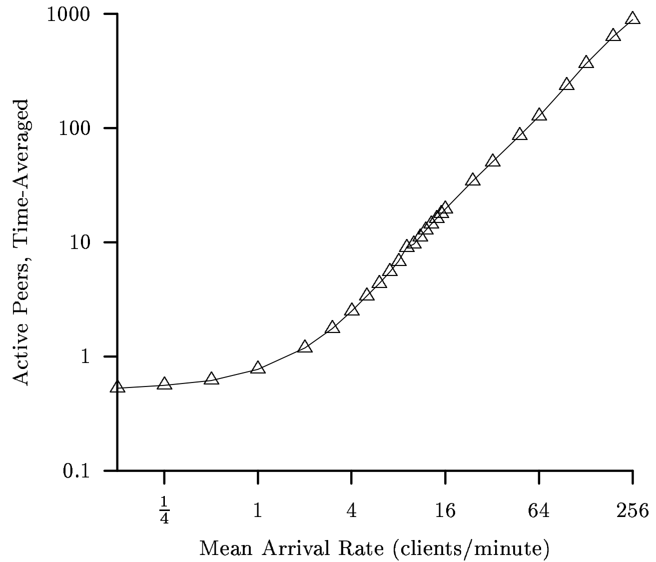

Fig. 10
Active Peers versus arrival rate

results of these simulations.

From these results, two trends are apparent. First, download efficiency decreases as the block size decreases. Recall that gossiping occurs before each transfer of a block. As the block size becomes smaller, the gossip consumes a larger percentage of the total throughput.

The second trend is as the block size becomes large, the download time increases slightly. This is a result of the "last block problem", which occurs when the last block to be downloaded is coming from a slow source. This causes the download to take a long time to fully complete, even if the most of the file was transferred quickly. Figure 14 illustrates the effects of the last block problem, using a one megabyte file with various block sizes. The y-axis displays the percentage of time transferring *only* the last block. In other words, this is the time spent waiting for the last block after all other blocks are transferred. My results indicate that for a $1MB$ file and a $256KB$ block size, the last block consumes 35% of the download time. BitTorrent [4] solves this problem by simultaneously downloading the last block from multiple sources. While this results in redundant data transmission, it potentially improves download times.

C.5 Client Distribution

With swarming, as with any peer-to-peer system, it is important to investigate the impact of low-capacity users on client performance. Some peer-to-peer protocols collapse when too many low-capacity users enter the system. For example, the original Gnutella protocol had this flaw. To address this concern, I conducted a variety of simulations using different mixtures of clients drawn from three classes: modem, broadband, and office. Table III lists the capacity of each class of users.

TABLE III Classes of Users

Type	Downstream	Upstream
Office	$43Mbps$	$43Mbps$
Broadband	$1536kbps$	$128kbps$
Modem	$56kbps$	$33kbps$

As parameters to the simulation, each class is assigned a probability. When a new client is created, it is given a class randomly, based on these probabilities. Other than the client capacities, the scenario is just like that described in Tables I and II. The arrival rate is 16 clients per minute.

Swarming appears to behave well when low-capacity clients interact with higher speed clients. Figure 15 shows the effects of modem users on broadband users. In this figure, only broadband and modem users are represented; thus, as the percentage of broadband users goes down, the percentage of modem users goes up. This figure shows that broadband users continue to obtain reasonable performance even as the mix of users is adjusted. As the percentage of modem users increases from 10% to 99%, the download time for broadband users increase by roughly a factor of two. While this

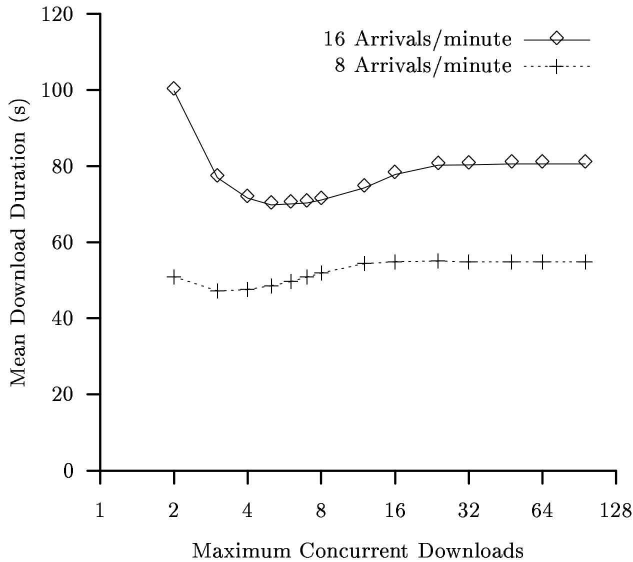

Fig. 11
IMPACT OF CONCURRENT DOWNLOADS

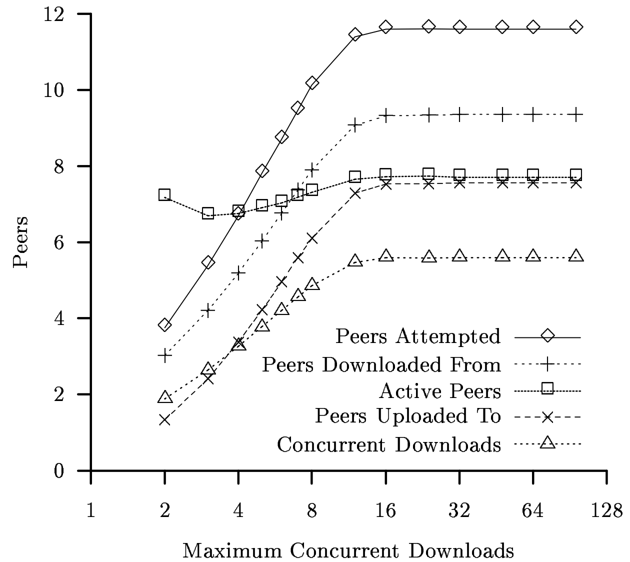

Fig. 12
PEER DYNAMICS FOR CONCURRENT DOWNLOADS

is a significant increase, the system clearly continues to function well despite an overwhelming number of low-capacity users. The performance of modem users is unchanged by large numbers of higher-speed users as their access link remains the bottleneck.

While I did get some results for office users, I did not run enough simulations with them to get a strong feel for the effect on them. However, my results show that inserting a small percentage of office users does not significantly improve the performance for broadband users. This is a side-effect of three aspects of my implementation. First, I conservatively assume that clients do not linger after completing the download. Thus, high capacity users do not stay around for long periods, helping slower users. Second, the last block problem described in Section VII-C.4 will limit gains from a small population of fast users. Finally, clients are not doing any kind of capacity-based peer selection. Introducing this mechanism could enable clients to take better advantage of friendly office users.

However, one open problem in peer-to-peer systems is *preventing* high-speed users from shouldering an unfair burden. Many deployed peer-to-peer applications allow end users to place a cap on the amount of capacity they dedicate to sharing. Relying on end users to allocate network capacity is not a good long-term solution.

VIII. Future Work

Swarming has excellent potential for future research. My research has demonstrated swarming's excellent scalability and flash crowd response. It has also begun to explore how and why swarming behaves as it does. Although conceptually simple, significant additional research will be needed to fully understand all of swarming's dynamics.

The next logical step is a systematic study exploring the importance of the four primary mechanisms of swarming, described in Section III:

- Peer Identification
- Peer Selection
- Data Division
- Data Selection

The goal of this study would be to gauge the effects of each mechanism. It should answer questions such as "What is the effect of a good Peer Selection algorithm versus a bad one?" This will help focus later studies on the areas with the largest delta.

For Peer Identification, the study would compare the performance of gossiping versus a centralized peer cache.[10] Intuitively, gossiping should do better under high load while a centralized peer cache would perform better under light load. However, the key questions remain "How high is high?" and "How much better is better?". Additionally, it would be worthwhile to study the effects of altering the contents of the tuples that are gossiped. For example, how would performance be affected if the block information was removed?[11]

[10] BitTorrent uses a centralized peer cache. [4]
[11] The Partial File Sharing Protocol used by Gnutella does this. [10].

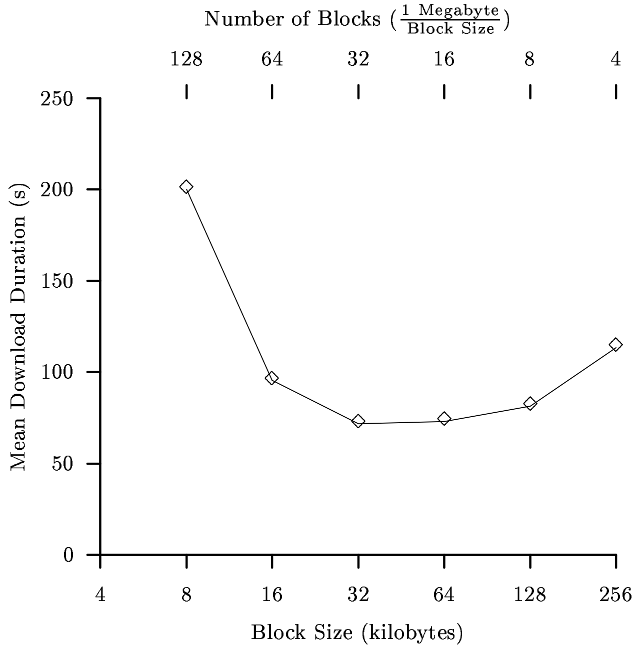

Fig. 13
EFFECTS OF BLOCK SIZE AND FILESIZE AT 16 CLIENTS PER MINUTE

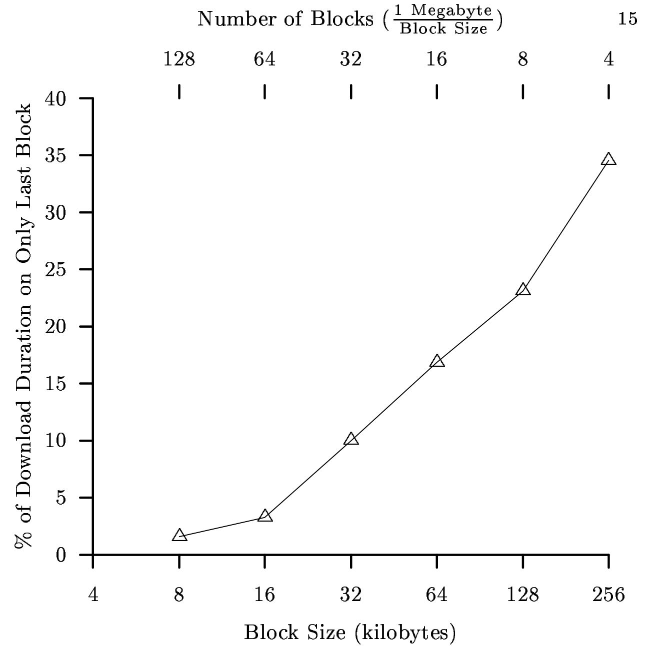

Fig. 14
SIGNIFICANCE OF THE LAST BLOCK PROBLEM

For Peer Selection, my current implementation chooses the peer with the largest number of useful blocks. The systematic study would compare this with a completely random algorithm. It would also include more intelligent algorithms, such as picking the least loaded peer. To simplify the task of building the simulator, some algorithms can take shortcuts by making use of global knowledge. In a real implementation, peers would need some way to transfer load information. In the simulator, it suffices to answer the question, "If we could do this, would it be useful?" In studying Peer Selection, the effects of the maximum concurrent peers[12] would also be studied under higher load.

The flip side of Peer Selection is deciding how many requests to accept. The study would explore limiting the number of concurrent uploads. In particular, this may reduce the heavy congestion at the server under high load. By adjusting this setting and treating the whole file as one block, it will be possible to roughly compare swarming with Pseudoserving [2] and CoopNet [1]. It would be interesting to see how this affects the flash crowd response.

For Data Division, the study would more closely examine the effects of altering the block size. In particular, it would include a graph similar to Figure 13, but with constant load. This would be done by decreasing the arrival rate as the filesize increases. Instead of using blocks, Data Division could be done with byte-granularity. This prevents unnecessary retransmissions when a connection is dropped part-way through a block. Rather than implementing byte-granularity, the study would simply measure how much retransmission occurs and compare it to the total amount of traffic. Finally, the study would examine the amount of time peers waste because they cannot find peers with useful blocks to serve. If this is significant, it would suggest that Forward Error Correction (FEC) might be useful.[13]

For Data Selection, the study would examine solutions to the last block problem, discussed in Section VII-C.4. This would include downloading the last block in parallel from multiple sources.

This project might also explore the effects of firewalled and non-participating users, depending on time constraints. It might also further explore the effects of different peer capacities.

A. Smaller Studies

There are a number of smaller studies that could be conducted. These might not be large enough for a paper of their own, but they would increase our understanding and might turn a good paper into a great paper.

One task is to gather measurements from real swarming implementations under heavy load conditions. This might be done using an artificial load using PlanetLab nodes [15], or it might be possible to collaborate with Bram Cohen, author of BitTorrent [4].

Another task is to create a high-level simulator for

[12] See Section VII-C.3

[13] This was used in SwarmCast [6]. Researchers have also used it in the client-server setting [9].

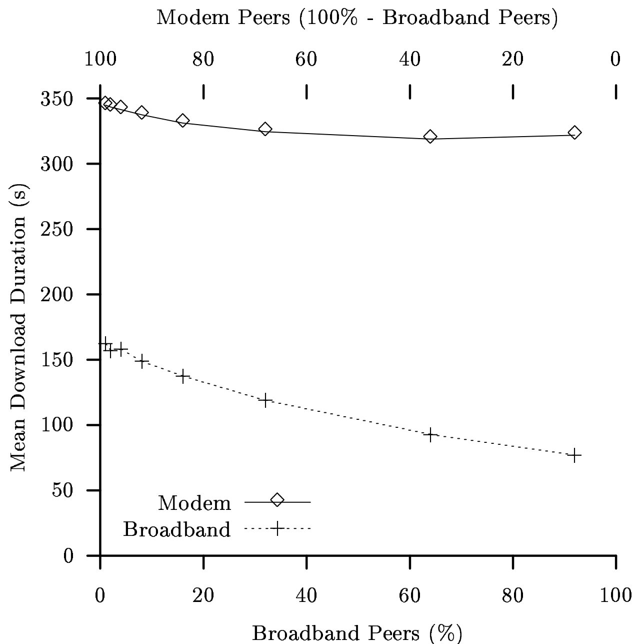

Fig. 15
Impact of low-capacity clients

visually observing the movement of blocks through the system. Using only a few blocks per file, it is easy to represent the blocks using colors. Consider a simple case of just 3 blocks in the file. If a peer has the first block, it will be red; if it has the second block, it will be green; if it has the third block, it will be blue. If a peer has the whole file, it is white. To simplify the simulation, all transfers of blocks are assumed to take unit time. It would be possible to simulate large networks using this system. Gossiping may have the flaw that popular blocks tend to become more popular, making some blocks difficult to find. This sort of problem would show up quickly with this visual simulator. In theory, it is simple to write.

IX. Acknowledgements

I would like to thank my DRP committee: Professor Daniel Zappala, Professor Ginnie Lo, and Professor Michal Young. Without their help my DRP would not have been possible. In particular, I would like to thank my research advisor, Professor Zappala, for his guidance, critiques, and insights which were helpful throughout my DRP. I would also like to thank Professor Reza Rejaie for his input near the end of the project.

References

[1] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai, "The Case for Cooperative Networking," in *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[2] Keith Kong and Dipak Ghosal, "Mitigating server-side congestion in the Internet through pseudoserving," *IEEE/ACM Transactions on Networking*, vol. 7, no. 4, pp. 530–544, Aug. 1999, `http://portal.acm.org/citation.cfm?doid=316739.316745`.

[3] Pablo Rodriguez, Andreas Kirpal, and Ernst Biersack, "Parallel-access for mirror sites in the internet," in *INFOCOM*, Tel Aviv, Israel, Mar. 2000, IEEE, pp. 864–873, `http://citeseer.nj.nec.com/rodriguez00parallelaccess.html`.

[4] Bram Cohen, "BitTorrent," `http://bitconjurer.org/BitTorrent`, 2003.

[5] "Open Content Network," http://www.open-content.net/.

[6] "Swarmcast technical overview," `http://web.archive.org/web/20010606012504/opencola.org/projects/swarmcast/swarm_tech1.shtml`, 2001.

[7] "eDonkey 2000," `http://www.edonkey2000.com/overview.html#how`, 2001.

[8] Daniel Stutzbach, Daniel Zappala, and Reza Rejaie, "Swarming: Scalable content delivery for the masses," In submission, 2003.

[9] John W. Byers, Michael Luby, and Michael Mitzenmacher, "Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads," in *INFOCOM*, New York, NY, Mar. 1999, IEEE, pp. 275–283, `http://citeseer.nj.nec.com/387283.html`.

[10] Tor Klingberg, "Partial file sharing protocl version 1.0," `http://groups.yahoo.com/group/the_gdf/files/Proposals/Working%20Proposals/PFSP/`, Aug. 2002.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2616, June 1999.

[12] Dimitri van Heesch, "Doxygen," `http://www.doxygen.org`, 2003.

[13] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc ACM SIGMOD Int. Conf. on Management of Data*, 1984, pp. 47–57.

[14] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu, "A Lightweight, Robust P2P System to Handle Flash Crowds," in *IEEE ICNP*, November 2002.

[15] "Planetlab," `http://www.planetlab.org`, 2003.