Swarming: Scalable Content Delivery for the Masses

Daniel Stutzbach
Computer and Information Science
University of Oregon
Eugene, Oregon 97403-1202
agthorr@cs.uoregon.edu

Daniel Zappala
Computer and Information Science
University of Oregon
Eugene, Oregon 97403-1202
zappala@cs.uoregon.edu

Reza Rejaie
Computer and Information Science
University of Oregon
Eugene, Oregon 97403-1202
reza@cs.uoregon.edu

***Abstract*— Due to the high cost of a Content Distribution Network, most Internet users are not able to scalably deliver content to large audiences. In this paper we study swarming, a scalable and economic content delivery mechanism that combines peer-to-peer networking with parallel download. First, we define a swarming architecture that generalizes the basic delivery mechanism in popular swarming protocols such as Gnutella and BitTorrent. We then conduct a comprehensive performance study of swarming delivery, using a variety of workloads. Our results show that swarming scales with offered load up to several orders of magnitude beyond what a basic web server can manage. Most impressively, swarming enables a web server to gracefully cope with a flash crowd, with minimal effect on client performance. During the course of our study we illustrate the benefits and limitations of a basic swarming protocol and identify several key opportunities for performance improvements.**

I. INTRODUCTION

One of the most compelling and unique aspects of the web as a communications medium is that any person has the potential to provide content to a global audience. However, the web has been only partly successful in realizing this ideal. Although the web has been incredibly successful with regard to *access* – for a small hosting fee, anyone can create a web site – the overwhelming majority of Internet users can not provide *scalable delivery* of their content to a large audience.

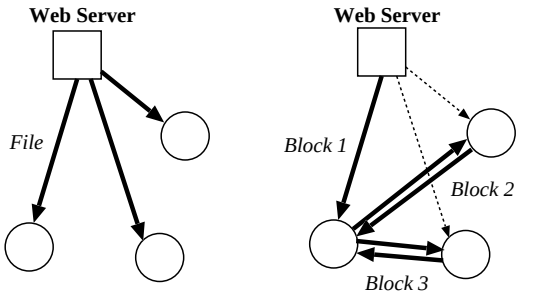
The main impediment to scalable content delivery is the web's dependence on a client-server model, which is inherently limited in its ability to scale to large numbers of clients. As the load on a web server increases, it must either begin refusing clients or else all clients will suffer from long download times. This makes it difficult for a website with limited bandwidth to serve large files or a large audience. Of particular concern in recent years is a *flash crowd* event, a phenomenon in which the client arrival rate at a web site grows by several orders of magnitude in a short time.[1]

The *masses* – ordinary users and small or medium-sized organizations – lack an effective means to deal with this scalability problem. Buying more bandwidth helps a site to serve a larger audience, but it takes a proportionately larger amount of bandwidth to serve a larger audience, and ultimately most users are limited in the amount they can pay. A more effective mechanism for dealing with this scaling limitation is a Content Distribution Network (CDN). A CDN improves scalability by distributing a given provider's content to a set of servers, splitting the load among them. The high cost of a CDN, however, makes this mechanism infeasible for all but the largest organizations. Another potential solution is proxy caching [2], [17], but this is useful primarily from the perspective of an individual client for whom the cache is available. From the perspective of the web server, caching must be deployed at a wide number of sites in order to be effective at reducing load. One last alternative is to multicast content from the web server to a group of clients [4], but this requires loose synchronization among the clients and mechanisms to accommodate heterogeneous client bandwidths. Moreover, multicast is not yet widely deployed.

Recently, peer-to-peer systems have emerged as an alternative paradigm for content delivery, addressing the scaling limitation of the client-server architecture by distributing the burden of content distribution among a large set of clients. Pure peer-to-peer applications, however, introduce two key problems: peer location and peer instability. With a web server, the location of the content is always known, whereas a peer-to-peer application must locate peers with the desired content. In addition, a web server typically stays connected to the network (unless of course it becomes overloaded), whereas in a peer-to-peer system peers may abruptly leave the network at any time.

In this paper, we study *swarming*, a peer-to-peer content delivery mechanism that utilizes parallel download among a mesh of cooperating peers. We integrate swarming with a standard web server to form a hybrid solution that combines the simplicity and stability of client-server delivery with the scaling benefits of a peer-to-peer network. For files that are small or not popular, the web server delivers content directly to clients (Figure 1a). However, as the popularity of a file increases, the server initiates swarming by giving clients only a block of the desired content, along with a list of peers that can provide other blocks of the same file. Swarming clients perform two basic functions. First, they gossip with their peers in order to progressively find other peers with the content they need. Second, as clients discover suitable peers, they begin to download the content from them in parallel (Figure 1b). Overall, the more loaded the web server becomes, the less content it serves directly to

[1]This phenomenon is also termed the *Slashdot Effect* because sites are often overrun with load when a story on the Slashdot web site links to an under-provisioned server.



(a) Client-server delivery: clients download the entire file from the server

(b) Swarming delivery: clients download blocks of the file in parallel from the server and peers

Fig. 1. Swarming Delivery

clients and the more it redirects them to peers. During heavy load, the system generates *swarms* of peers that cooperatively download content in parallel from each other and from the server.

Swarming is a viable content delivery mechanism for the masses because it is both scalable and economical. Swarming actually uses scale to its advantage – system capacity increases with the number of peers participating in the system. Peers spread the load of content delivery over the entire network and share the burden of peer identification with the web server; this prevents server overload and avoids network congestion. Clients utilize parallel download to protect themselves against peer instability, which would otherwise hinder a peer-to-peer application. Swarming is clearly an economical solution for the web server because it does not have to pay for the bandwidth used for peer-to-peer communication. This cost is marginal for the peers because each peer serves only a small number of clients; the server's cost is spread among a large user population. In addition, clients have an incentive to participate in swarming because they will receive the help of other peers in return (for the same or different content). This improves the average performance of all users that otherwise would suffer from the congested server or network.

While we advocate swarming as a standalone solution, we note that swarming is complementary to both CDNs and proxies. Swarming can enable a CDN server to handle a higher load, and proxies can use swarming to reduce the load on a standard web server. Furthermore, organizations that don't use a CDN can utilize swarming as an alternative to provisioning their network for peak load.

Swarming delivery has been popularized by several proprietary and open-source software projects [1], [12], [8], [21], [7], of which BitTorrent is perhaps the most well known. While these systems serve as a proof-of-concept for swarming delivery, few provide a technical description of their swarming protocol and, more importantly, no performance evaluation 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. Some challenges include (a) finding peers with the desired content, (b) choosing peers that are likely to provide good performance, and (c) managing parallel download while coping with partially available content at each peer and the dynamics of peer participation.

In this paper, we make the following contributions. First, we present a comprehensive swarming architecture and explore the design space of its key components. Second, we conduct the first comprehensive performance evaluation of swarming delivery, using a simulation that examines a variety of workloads and swarming parameters. Our results show that swarming can scalably deliver content under loads that are several orders of magnitude beyond what a client-server architecture can handle. Most impressively, swarming enables a web server to gracefully cope with a flash crowd, with minimal effect on client performance. In addition, our results indicate that swarming spreads the load of content delivery evenly among the peers. We conclude by providing insight concerning the dynamic performance of the system and the impact of several key swarming parameters.

II. RELATED SYSTEMS

BitTorrent is notable as a swarming system because it is currently used to transfer large files, such as new software releases, to hundreds of peers. With BitTorrent, a centralized host called a *tracker* is responsible for storing the identities of all peers and their performance. When clients contact the tracker, they report their status and in return receive a random list of peers. Clients try to download "rare" blocks first (based on the blocks their peers have) and download one block at a time by requesting sub-blocks of their current block from selected peers. Once a client obtains a complete block, it can share that block with its peers.

One of the unique features of BitTorrent is its notion of fairness. Each connection between peers represents a two-way data transfer, with each peer expected to upload as much as it downloads. At any given time, each client allows a fixed number of connections to be actively uploading, with the goal of obtaining good download performance from those same peers. If a given peer does not provide good download performance (i.e. the other side is not sharing equally), then the client will stop uploading to that peer and try a different peer.

A number of other peer-to-peer systems attempt to address the same problem of serving web content to a large audience. The systems that are most related to swarming are CoopNet [13] and Pseudoserving [10]. Both of these systems use collaborative delivery – an overloaded web server gives clients a list of possible peers, and the client chooses a single peer from which it downloads the entire content. CoopNet in particular provides a proof-of-concept for collaborative delivery that serves as a foundation for swarming. First, the authors make a convincing argument that bottleneck bandwidth at the server, rather than the server CPU or disk speed, is the limiting factor in client-server content distribution. Second, this work demonstrates that clients are able to find content using the list of peers, that load is distributed across the peers, and that clients can find peers that are "close" (in the same BGP prefix cluster). Finally,

CoopNet has also been extended to deliver streaming content [14].

Swarming differs from these systems in several important ways. First, it uses parallel download, which balances the load among peers and provides robustness against peers that leave the system. Second, swarming allows clients to act as peers even if they only have partial content, which further increases system capacity. This also allows swarming to respond more quickly to a flash crowd, especially for large files. Finally, swarming uses gossiping to progressively discover available peers, which better distributes control overhead.

The Backslash system [18] helps a web server cope with high load by forming a collaborative network of web mirrors. An overloaded web server then redirects clients to a cached copy of the content located at one of the collaborating sites. While this type of system is economical and can improve the ability of a web server to respond to high loads, it scales with the number of participating servers, whereas swarming scales with the number of clients. Moreover, the network of cooperating sites must be established ahead of time and benefits only the participating servers.

PROOFS [19] uses a peer-to-peer network of clients to cache popular content. When a client is unable to download content from a web server, it queries the peer-to-peer network to see if any other user has a copy of the desired content. Like swarming, this type of system scales with the number of participating clients. However, the peer-to-peer network does not prevent the web server or the network from becoming overloaded; rather it serves as a backup after a web server (or the network) becomes congested. This approach is thus complimentary to any other content-delivery system, including swarming.

All of these systems are affordable alternatives for the sites that cannot pay for a Content Distribution Network (CDN). However, even for sites that can afford a CDN, we argue that swarming provides some advantages. From a content provider's point of view, swarming provides automatic and dynamic content management, whereas a CDN needs additional mechanisms to manage replication and consistency. Swarming also has the potential for better load balancing, due to parallel download. When swarming uses proximity-based peer selection, it has the potential to further reduce network load by serving content from peers that are likely to be much closer than a CDN server. Finally, where a CDN is available, swarming is complementary in that users can swarm to the set of CDN mirrors, further improving the scalability of the overall system.

Our design of swarming draws on two well-known techniques – parallel download and gossiping. Downloading from multiple web mirrors in parallel [3], [16] has been shown to reduce client download time while also spreading load among the mirrors. We borrow this technique to allow clients to download content from multiple peers in parallel. While the concept is similar, for swarming the peers may have only partial content, may disconnect abruptly, and are not as fully provisioned as a dedicated server. These complications add significantly to the dynamics of swarming compared to downloading from mirrors. The second well-known technique – gossiping – has primarily been used to maintain consistency, for example in distributed databases [6], [9] and Peer-to-Peer storage networks [5]. For swarming we use gossiping as a scalable method for a client to explore existing peers in a demand-driven fashion.

III. SWARMING ARCHITECTURE

In order to study swarming performance, we have designed a swarming architecture consisting of four key components: *swarming initiation*, *peer identification*, *peer selection*, and *parallel download*. While we have not made an explicit attempt to base this architecture on any existing swarming protocol, we believe it generalizes the basic content delivery mechanism used by protocols such as Gnutella and BitTorrent.

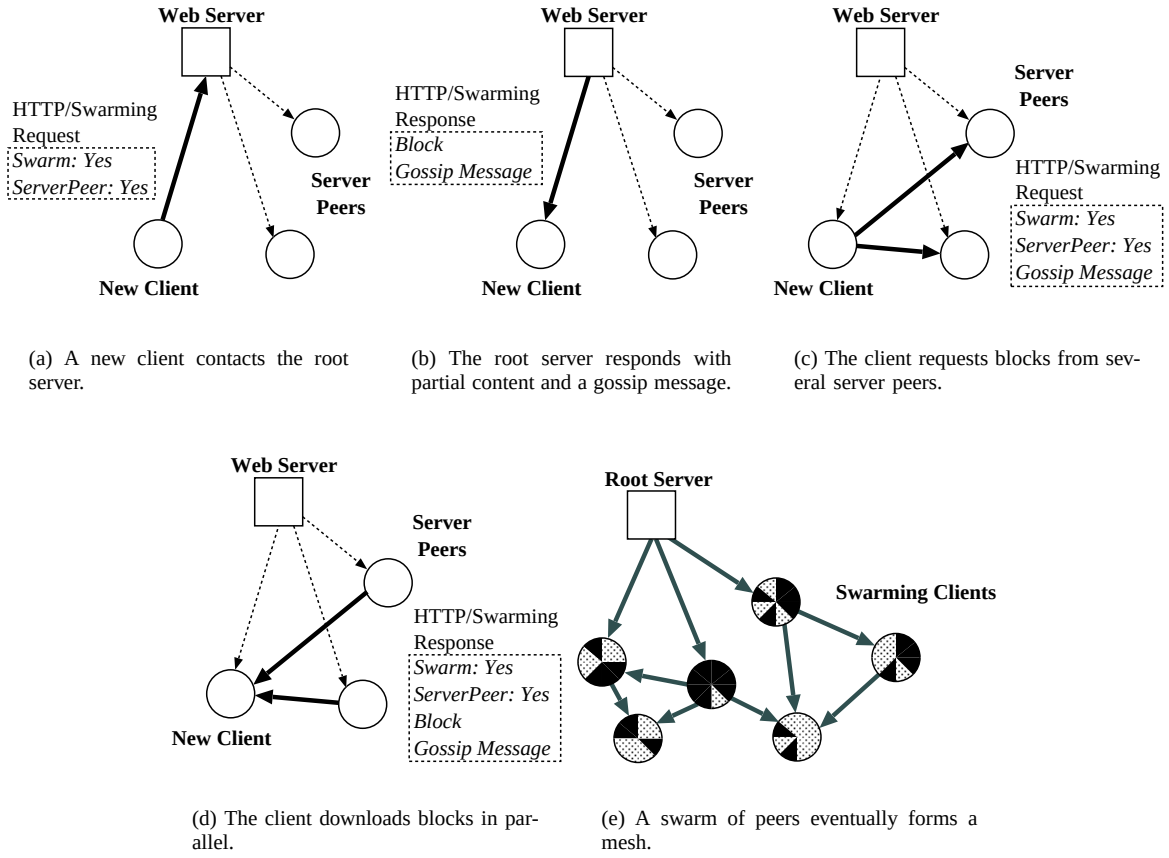
We note that there are many design choices for each swarming component, as well as many parameters for the system. Our goal is to use a simple yet effective design for each component. This enables us to study the performance of swarming delivery while minimizing complex dynamics and interactions among the components of the system. This makes it easier to correlate an observed behavior to a particular mechanism or parameter.

Before describing each component in detail, we provide an overview of the swarming architecture. Our integration of swarming with a web server can be viewed as a hybrid between client-server and peer-to-peer content distribution. The system uses client-server communication to deliver small or unpopular files, to bootstrap peer location, and to serve as a fallback in case a client's known peers all leave the network. The system uses peer-to-peer networking to scalably deliver large or popular files and to discover additional peers through gossiping.

In our architecture we describe swarming as a protocol that is implemented on top of HTTP, providing backward compatibility and allowing for incremental deployment. Because peers can act as both clients and servers, we define some basic terminology. We use the term *root server* to refer to the server that is the content owner. This could be a regular HTTP server or a CDN mirror. We refer to a client as a *client peer* when it acts as a client and a *server peer* when it acts as a server. To participate as a server peer, a node runs a lightweight HTTP server. It is important to note that the client may be either a web browser or proxy server. It should be simple to integrate swarming into existing proxies because they already include server functionality.

A. Overview

Swarming clients send regular HTTP requests to web servers, along with two additional headers. The SWARM header indicates that a client is willing to use swarming, and the SERVER PEER header indicates that a client is willing to act as a server peer for the requested file (Figure 2(a)). For clients that are willing to swarm, the root server may respond either with the entire file – when load is light – or



(a) A new client contacts the root server.

(b) The root server responds with partial content and a gossip message.

(c) The client requests blocks from several server peers.

(d) The client downloads blocks in parallel.

(e) A swarm of peers eventually forms a mesh.

Fig. 2. Protocol Example

may initiate swarming if needed. Servers that are not capable of swarming will simply ignore the unrecognized headers.

To initiate swarming, a root server gives some clients a single *block* of the file and a *Gossip Message* (Figure 2(b)). A block is a portion of the file, typically on the order of tens or hundreds of kilobytes. The server determines the block size on a per-file basis; our performance evaluation looks at a range of file and block sizes. A gossip message contains a list of peers that are willing to serve portions of the same file. For each server peer, the message lists the peer's IP address, a list of blocks the peer is known to have, and a time stamp indicating the freshness of this information.

When a client receives a swarming response (partial content plus a gossip message), it invokes a *peer selection* strategy to determine the subset of server peers from which it will download content. A client's primary concern is to locate blocks of the file that it has not yet received. The client then begins downloading blocks from both the root server and the server peers in parallel. We note that in this study we are not concerned with fairness. Thus, unlike BitTorrent, data transfer over a connection is one-way, and we do not attempt to ensure that a client downloads only as much as it uploads.

Each transaction between a client and the root server or a server peer includes a single block and a two-way exchange of gossip messages (Figure 2(c)-(d)). Requesting a single block at a time will naturally lead to faster peers delivering more blocks, resulting in proportional load balancing. Exchanging gossip messages with server peers enables progressive *peer identification* – clients gradually learn about other peers in the peer-to-peer network. This is needed because the initial pool of peers identified by the root server may refuse to serve the client, may disconnect from the network, may have low bandwidth connectivity, or may simply not have all of the content the client needs. The client uses gossiping to distribute the overhead of peer identification, rather than relying on the root server for this functionality. In addition, since a client is conducting a transaction with a server peer anyway (to download a block), the additional overhead of including a gossip message in the transaction is marginal.

When a server peer receives a request for a block, it determines whether it will accept the connection based on its configuration or capabilities. The server peer then delivers the requested block and exchanges gossip messages with the client. Once the server peer has itself downloaded the entire file, it may decide to leave the system immediately or it may choose to linger and help additional client peers. In our study we use a *lingering time* of zero, in order to test swarming under pessimistic conditions; in practice the user may specify a lingering time or swarming may continue as long as the browser is left open. When a server peer disconnects from the system, it does not wait for any ongoing block downloads to finish. To reduce the amount of bookkeeping that is required,

clients discard partially downloaded blocks.

As large numbers of clients attempt to download the same content, they form a dynamic mesh or swarm of peers. We can view this mesh as a *collaborative delivery system*, where server peers with larger portions of the file or higher bandwidth will tend to serve greater numbers of clients. We illustrate this in Figure 2(e) by depicting the download process at each node as a pie with pieces that are being filled. Downstream peers will generally get pieces from upstream peers who have received the content earlier. Similar to application-layer multicast, collaborative delivery spreads the load of transferring content among the clients and eliminates bottlenecks at the source or other points in the network. We note that while cycles may form in the delivery mesh, individual blocks are propagated along a tree that starts at the root server.

B. Swarming Initiation

A root server needs to decide when to initiate (or cease) swarming for a particular file based on its current performance and the popularity of the file. Moreover, while swarming, the server needs to decide what portion of clients to give just a single block and how many to serve with the entire file. The server must balance its desire to reduce load via redirection with the need to keep enough content available for swarming delivery to be effective.

For our performance evaluation we have chosen the conservative approach of swarming at all times. This enables the root server to be proactive with regard to load, so that it doesn't react too late to a sudden increase in client arrivals. During a flash crowd, it is conceivable that load on the server (and its access link) can increase several orders of magnitude, swamping a server that is slow to respond. The cost of swarming at all times is that when load is low clients see increased delay compared to downloading the entire content from the root server. This occurs because clients may download from peers that are slower than the root server.

In keeping with this conservative approach, we also have the root server send a swarming response (partial content plus a gossip message) to all clients. This represents a balance between giving clients too little or too much. During high load, it may be desirable to only give clients a gossip message, because this will reduce the load on the server to just redirection overhead. On the other hand, this runs the risk of not giving the clients enough to share, at which point they must return to the root server for content anyway. Another alternative is to give some clients the entire file so that they can act as server peers for all blocks. However, this benefit is lost if the lingering time is very short, or in other words if the peers act selfishly. By giving all clients only partial content, we force them to cooperate with each other. The content will naturally diffuse as clients exchange blocks with each other and with the root server.

C. Peer Identification

A client needs to locate other peers who have its desired content so that it can use them as server peers. Identifying potential peers is difficult because the set of peers interested in the same content is not known ahead of time and can be highly dynamic (due to client disconnections). This means we cannot use a distributed hash table [20], [15], [11] to locate content, but must instead use a more dynamic peer discovery mechanism. Fortunately, a client does not need to know about all peers with the content; instead, a client needs only a few peers with whom it can perform swarming.

For our performance evaluation we use a combination of server-based identification and gossiping. Having the root server supply an initial set of peers is a simple method for bootstrapping the peer identification process. However, we do not want to rely on this mechanism for all peer identification (as is done with BitTorrent, CoopNet, and Pseudoserving) because in a very large scale system even this redirection load may overwhelm the server. Instead, clients and server peers gossip during each transaction, allowing clients to quickly discover a small fraction of the peers and their available content. This mechanism provides both scalability – the number of peers discovered is small relative to the total number – and robustness – a failure or disconnection of one peer does not affect the ability of a client to discover other peers. We also note that finding suitable peers becomes easier (and hence consumes less overhead) as the number of active peers increases. Of course, if a client is unable to find suitable peers, it may always return to the server to ask for additional peers. Finally, it is easy to limit the overhead of gossiping by limiting the frequency with which nodes exchange messages and by limiting the size of the gossip message.

Our emphasis in designing the gossip component is to discover recent peers, since peers may leave the system at any time. We consider the dynamics of peer participation to be our primary challenge, even more important than optimizing bandwidth. Hence, we specify that each client caches a record for the N_c peers with the most recent time stamp, then includes in its gossip messages the most recent N_g peers, where $N_g \leq N_c$. In keeping with our philosophy, freshness takes priority over other concerns (such as caching peers with large numbers of blocks), since a peer with the entire file is not useful if it leaves the system. Moreover, the client has an incentive to exchange fresh peers to ensure that gossiping diffuses information about peers effectively.

In our design of the gossip component, we are also careful to share information about peers that are no longer available. If a client does not do this, then the bad information effectively "pollutes" the peer identification mechanism, causing many peers to attempt to contact the same disconnected peer. In our study a client indicates a peer is disconnected by modifying the peer's gossip record to indicate the peer does not have any blocks of the file. The client then shares this record in its gossip transactions, so that other clients do not attempt to contact this peer. Eventually the record of a disconnected peer is discarded because its time stamp will never be renewed.

Finally, we note that our design uses *passive gossiping*, in which clients exchanges gossip messages with server peers during each transaction. An alternative is to use *active*

gossiping, which requires the client to choose a gossip frequency and then continually contact a new peer during each round. While information diffuses more slowly with passive gossiping, the overhead is also much lower, since the amount of information in the gossip message is usually small compared to the data that is exchanged.

D. Peer Selection

Once a client has located potential peers, it needs to decide which peers and how many peers it should use for parallel download. These are difficult choices because the client does not know ahead of time the average bandwidth available from each server peer. In particular, the client does not know if the bottleneck of the connection will be local or remote.

We do not focus in this study on an optimal peer selection strategy, since there are so many other factors that affect swarming performance. Rather, we formulate a simple strategy that is based on content availability. First, each client limits itself to N_d concurrent downloads. Second, when choosing a new peer, clients choose the peer that has the most blocks that it still needs. 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.

We use this simple strategy because the most important factor in selecting a peer is the available content at that peer. This is particularly important for swarming because each peer has only a part of the file; a client *must* select among peers that have blocks the client is currently lacking. It only makes sense to consider other criteria – such as distance or available bandwidth – if multiple peers can provide the same content.

When choosing the limit of N_d parallel downloads, clients must balance several factors. If the client uses a small number of peers, then it may not fully utilize its incoming link capacity. If the client uses a large number of peers, then the extra peers may not necessarily improve performance, due to a bottleneck near the client. In this latter case, block download times will increase; due to the instability of peers, this in turn increases the probability of downloading incomplete blocks. Because we discard incomplete blocks, this results in useless work and reduces performance. We study a range of settings for N_d to determine the impact of this parameter on performance.

E. Parallel Download

The heart of the swarming architecture is the parallel download of different blocks from server peers. While swarming, the client must deal with long-term dynamics and must determine when to add or drop a peer. In addition, because the client is using parallel download, it must decide which blocks to download from which server peer, while coping with the fact that each peer may potentially have a different set of blocks.

In our study, we use a relatively simple strategy for adaptive delivery in order to simplify the analysis of our results. Each client chooses N_d peers for parallel download, using the peer selection component, then continues to use this set unless a server peer disconnects or runs out of blocks that the client needs. In either of these cases, the client drops the peer and then immediately invokes the peer selection component to choose a replacement. If none of the peers in the client's gossip cache have blocks that the client needs, then the client contacts the root server for some additional peers.

In keeping with our goal of simplicity, we do not monitor the performance of a server peer to determine whether to continue using it. Because this could lead to instability, we instead rely on the benefit of parallel download – faster peers will naturally serve more blocks to a given client. We also do not enforce any limit on the number of clients that can use a particular server peer. Our results indicate that our simple delivery mechanism spreads load evenly among peers, so overload of a given peer is not yet a concern. This is likely to be more important if clients begin using more sophisticated peer selection mechanisms.

While downloading content, the client would like to keep its current peers busy to ensure that its throughput is high. In our study we try to ensure that this is the case by having the root server choose a relatively large block size. We note that this choice has several drawbacks. Using too large of a block size will reduce performance because it increases the chance of a client getting a partially-downloaded block. We study the effect of block size in our performance evaluation.

Finally, we note that it is important for the root server to ensure that a variety of blocks are diffused to clients. If all clients have the same blocks, then they will all need to return to the root server for additional data. To mitigate this concern, in our performance study a client selects a block randomly from those available when connecting to both the root server and server peers. This ensures some amount of diversity in the content that is available, and increases the chance that a client can find a peer with content that it needs.

IV. PERFORMANCE EVALUATION

We have conducted a simulation-based evaluation of swarming to study its performance under a variety of workloads. Our simulation implements a swarming protocol that follows the architecture described in the previous section. In particular, swarming initiation is conservative – swarming is enabled at all times and the root server sends at least one block to each client. This first choice ensures that swarming is able to react to a flash crowd when it appears; the second choice ensures that the clients have enough content to share with each other. Peer identification, both by the root server and through gossiping, is based on freshness and peer selection is based only on available content. We use a simple adaptive delivery component in order to avoid introducing further dynamics in the system; a client stops using a peer only if it disconnects or runs out of blocks that it needs.

We first describe our simulation methodology. We then begin our study by demonstrating the scalability of swarming as compared to a standard web server under a steady-state

Parameter	Value
N_d (Concurrent downloads)	4
N_c (Size of gossip cache)	64
N_g (Peers in gossip message)	10
Block Size	32 KB

TABLE I
DEFAULT SWARMING PARAMETERS

Parameter	Value
File size	1 Megabyte
Server bandwidth	$1Mbps$
Client bandwidth (down/up)	$1536Kbps$ / $128Kbps$

TABLE II
BASIC SWARMING SCENARIO

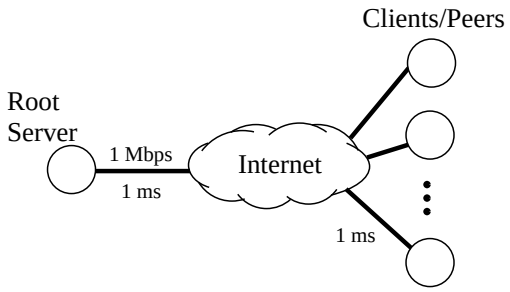


Fig. 3. Simulation Topology

load. Following this, we illustrate how swarming enables a web server to handle a flash crowd without a significant performance hit. Next, we examine the performance of swarming in detail under high load, then study the dynamics of peer selection. We conclude by investigating the impact of a variety of parameters: file size, block size, number of concurrent downloads, and client distribution.

Unless otherwise mentioned, we use the default swarming parameters given in Table I.

A. Methodology

Evaluating the performance of swarming can be complex because of the many dynamics involved, such as peer participation, partially available content, and changes in available bandwidth. Moreover, the components of a swarming protocol are inter-related; for example, the information carried in gossip messages affects the peer selection component, which in turn affects the performance of adaptive delivery. Where possible we try to study the effect of a parameter in isolation, for example we investigate the effect of client bandwidth without dealing with network congestion in the backbone.

For this study we use a simulator based upon some of the original *ns 1.4* code. We built an HTTP server and client on top of TCP, with swarming integrated into both the server and client. We have tuned this simulator for scalability, since we need to evaluate swarming under extremely high loads.

1) Topology: Similar to congestion control studies, we use a simplified topology in which we model the Internet as a single router, as shown in Figure 3. This abstraction enables us to focus on the bottlenecks at the root server and client peers. The server's access link is likely to be a bottleneck under high loads [13] – the very loads for which we are designing swarming – and peer links are likely to be the bottleneck when a peer with limited bandwidth acts as a server peer.

Most of our simulations use the basic scenario shown in Table II, in which the root server has a 1 Mbps access link and serves a 1 MB file. In most simulations we model the clients as broadband users, with a download bandwidth of 1536 Kbps and an upload bandwidth of 128 Kbps. Using a higher download bandwidth is interesting because this makes parallel download attractive for a client. In addition, using a homogeneous set of clients allows us to focus on other dynamics in the system. In the latter part of this section we explore the effects of adding some low-bandwidth and some high-bandwidth clients into the system. In order to focus on transmission delay, we set the propagation delay of all links to 1 *ms*.

2) Workload: We control workloads for our simulations by varying the arrival rate of clients requesting the same file from the root server. For a given arrival rate, we randomly generate client inter-arrival times using an exponential distribution. We also simulate a flash crowd by abruptly increasing the arrival rate for a given period of time.

When a client arrival occurs, we create a new client and it immediately begins its download. During the download, the client acts as a server peer, then it leaves the system once its download is complete. While in the real world clients may be somewhat more polite, we opt for a conservative approach and hence underestimate the benefits of swarming.

Another key factor in determining workload is the file size. We study various file sizes as well as various block sizes (given a fixed file size) to determine the effect of these parameters on system performance.

3) Metrics: Our primary performance metric is client download time. We also measure the packet loss rate, the number of clients served by each peer, the number of blocks served by each peer, and a variety of other swarming-related metrics.

Unless otherwise indicated, we begin each simulation with a warm-up period of 500 download completions. During this time we do not collect measurements; this allows the system to reach steady state behavior. We then collect data for 5500 download completions.

For each experiment we conduct multiple runs of our simulations, average the results, and compute the 95% confidence interval. We do not include confidence intervals here because in all cases they are very small.

B. Scalability

We begin our study by showing that swarming has excellent scalability. In Figure 4, we plot the mean time a client takes to fully download a file versus the client arrival rate on a log-log scale. Swarming can clearly handle a much larger load than a basic web server. As the load increases, swarming exhibits a linear increase in delay, whereas client-

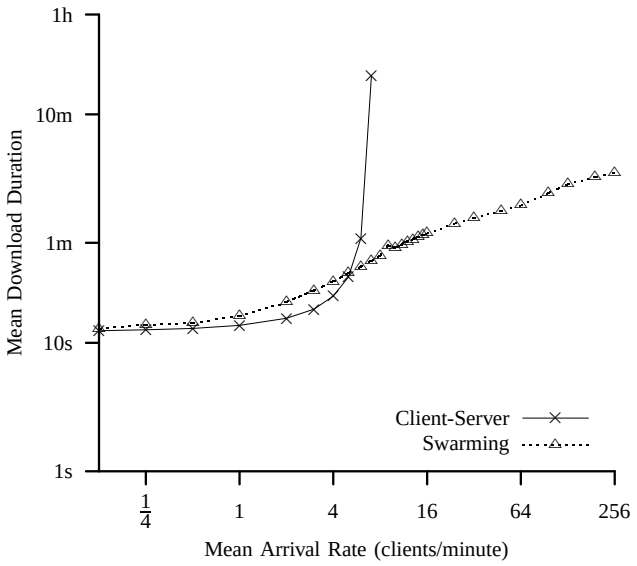


Fig. 4. Impact of arrival rate on performance

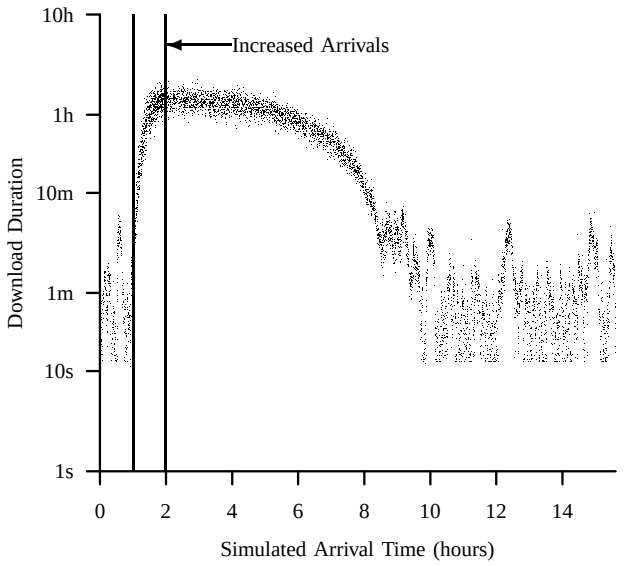


Fig. 5. Client-server reaction to a flash crowd

server transfer experiences super-exponential growth. Client-server has a vertical asymptote at about 7 clients per minute, beyond which it utterly fails to handle the load. Past this point, the arrival rate exceeds the departure rate, and the client download time continues to increase indefinitely. Naturally, the point at which the client-server protocol is unable to respond will depend on server bandwidth, file size, and load.

We were unable to find any bound for swarming, due to memory limitations (the largest arrival rate we are able to simulate is 192 clients per minute). Inevitably, an asymptotic bound for swarming must exist; 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.[2] This limitation 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 [19].

From this result we can see that swarming dramatically increases the steady-state load that a web server can handle. Serving 192 clients per minute translates to 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![3]

Our conservative approach to 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 that we can eliminate this problem by designing a dynamic server initiation component that uses swarming only when needed.

C. Flash Crowd

While good steady-state behavior is important, web servers must also be able to cope with extreme bursts of activity called flash crowds. We simulate the effect of a flash crowd by abruptly increasing the arrival rate for a fixed period of time. We begin by using a one-hour steady state load of 6 clients per minute. For client-server transfer we introduce an impulse of 12 clients per minute, lasting for one hour. After the flash crowd passes, the arrival rate returns to its original level, and we simulate this load until the web server is able to recover. For swarming, we provide a more challenging flash crowd by increasing the flash crowd rate to 120 clients per minute![4] Aside from the load function, we use the same swarming scenario given in Table II. The results are presented in Figure 5 and Figure 6, where each data point represents the mean download time for all downloads finishing in the previous 1000 seconds.

As can be seen from these figures, swarming enables a web server to smoothly handle large 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. With the traditional client-server approach, the crowd swells due to an inability to service the requests. This causes a death spiral, as the larger the crowd, the more difficult it is to service any requests at all. The server will not recover until long after the arrival rate decreases.

It is particularly impressive that we achieve this result using a conservative and unoptimized swarming protocol as

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

[3]`http://www.bandwidthsavings.com/servicesdetail.cfm`

[4]In order to fill out the graph, we ran this simulation for 12,000 completions instead of the usual 6000.

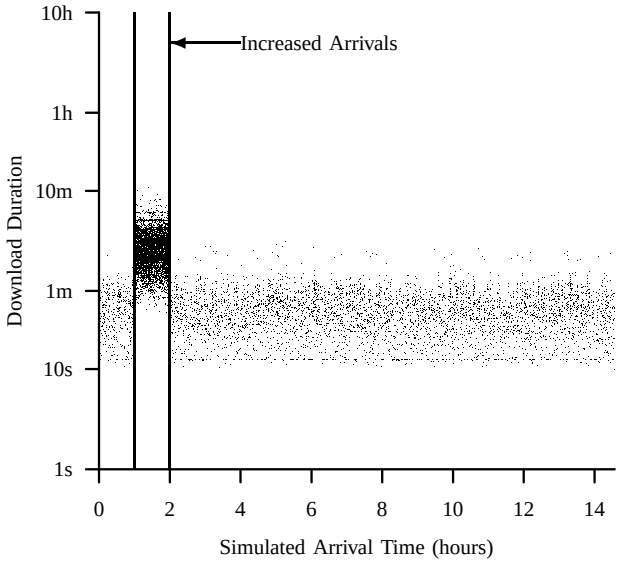


Fig. 6. Swarming reaction to a flash crowd

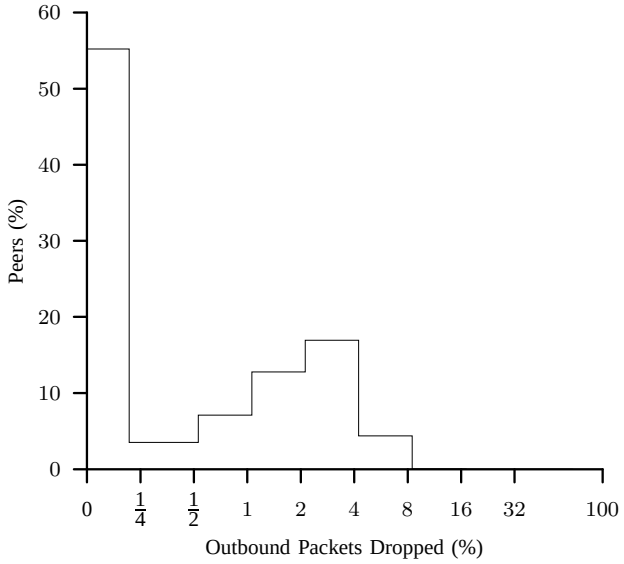


Fig. 7. Histogram of packet loss rates - 192 clients per minute

detailed in Section III. In particular, the simulated protocol has swarming enabled at all times and the server continually delivers blocks to clients as requested. With additional optimization, particularly in the swarming initiation component, the server should be able to handle even larger flash crowds. In fact, by utilizing Gnutella or PROOFS to locate peers under extremely high loads, *swarming can be made effectively immune to flash crowds*.

D. High Load

Now that we have shown that swarming can provide greatly increased system capacity, we must examine what sort of burden it places on the server and the peers. We focus on a very high load of 192 clients per minute, since this is the region where packet loss and load imbalance can be the worst. We again use the scenario from Table II.

With swarming at this heavy load, packet loss at the root server is quite severe. We stress that the client-server protocol has virtually 100% packet loss with an order of magnitude less load. Impressively, swarming still manages to get the file delivered to clients. Even at this high load, the congestion at the root server can be relieved by using a server initiation component that only delivers blocks of the file to a subset of the clients. This would allow the server to primarily perform redirection, while serving enough content to ensure it is available to the clients.

Unlike the server, the peers experience very little packet loss, even at high load. This is illustrated by the histogram of peer packet loss rates shown in Figure 7, using a logarithmic scale.[5]

The low packet loss rates at the peers can be attributed 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 8, which plots a cumulative distribution function megabytes served by peers. Even if a peer has served a whole megabyte, it may not have served the whole file, since it may simply have 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.

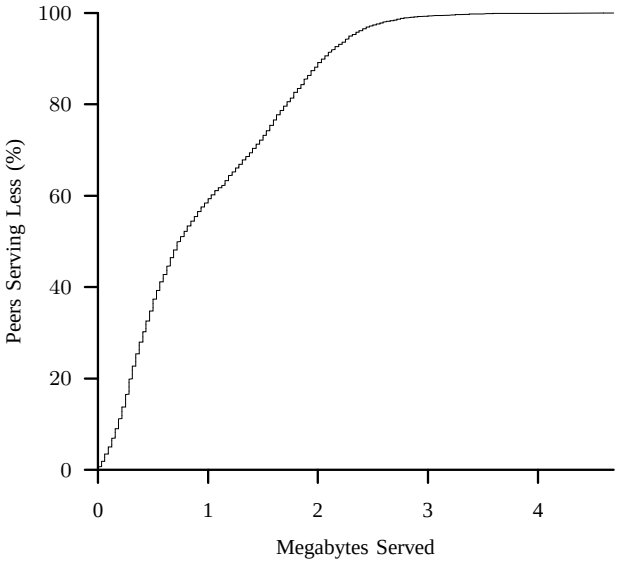


Fig. 8. CDF of megabytes served at 192 clients per minute

The time for a client to complete its download is less evenly distributed than the amount of data served. For the high load scenario, the download times are spread mostly over a range between 60 seconds and 300 seconds. However, more notably, a disproportionate number of download times are close to multiples of 60 seconds. This is shown as a

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

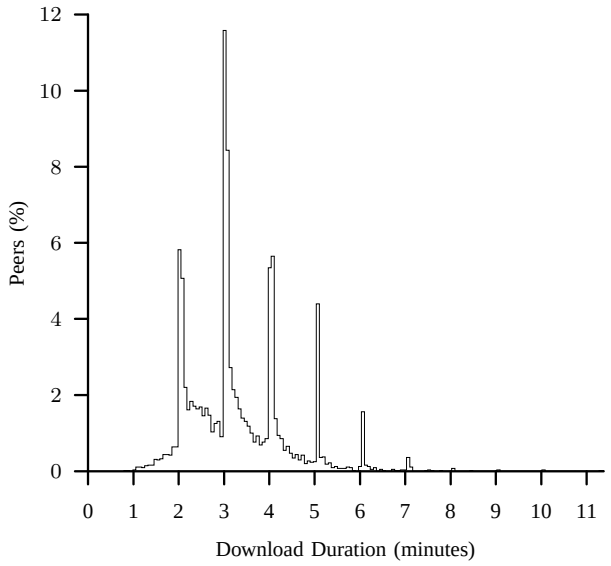


Fig. 9. Histogram of download times at 192 arrivals per minute

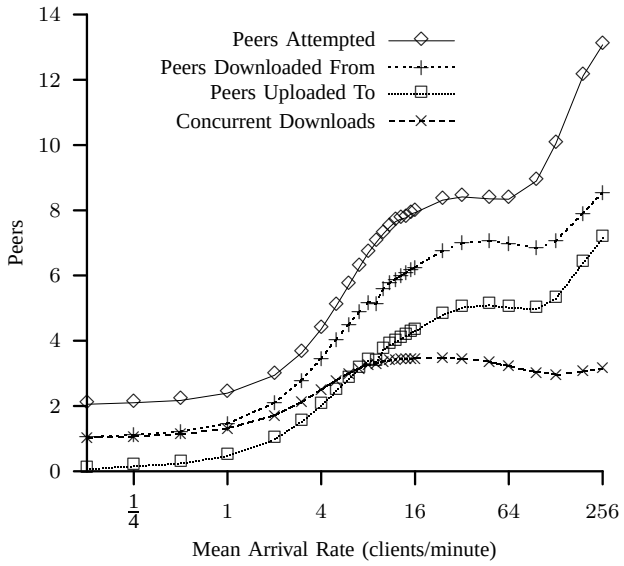


Fig. 10. Dynamics of Peer Selection

histogram in Figure 9. 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, we were able to confirm that the uneven distribution is caused by our use of a 60 second time out 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.

E. Dynamics of Peer Selection

To better understand the dynamics of peer selection, we examine several peer-related metrics under various loads. We are interested in the number of concurrent downloads that a client is able to perform, the number of unique peers that a client downloads from, the number of unique peers that a client serves, and the total number of peers that a client attempts to contact. Figure 10 plots these metrics as a function of increasing load, once again using the basic scenario given in Table II.

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-averaged, so for example if a client spends half the simulation downloading from 3 peers and half of it downloading from 4, then the average for that peer is 3.5.

This result also shows that the number of peers a client attempts to contact increases as the load on the system increases. At higher loads there are more active peers in the system, but each peer is downloading at a slower rate. This means that a client will need to contact more peers to find the blocks it needs. Accordingly, the number of peers a client downloads from increases during the region of high load. Likewise, the number of peers a client uploads to shows the same behavior.

F. File size

Swarming also scales well with the size of the file, allowing a small user to easily serve large files (e.g. multimedia). We demonstrate this result in Figure 11, which shows the mean download time for both swarming and client-server as a function of the file size. For this simulation we use an arrival rate of 4 clients per minute, with the same basic scenario given in Table II. Varying the file size is similar to varying the arrival rate in that both cases increase the load on the root server. Swarming again exhibits only a linear performance hit under high load (large files), and for file sizes of two megabytes or larger the client-server protocol is unable to enter steady-state.

An interesting result from this simulation is that gossiping can impose a significant overhead when the block size is small. For this simulation the number of blocks is 32, regardless of file size. Thus as the file size decreases, the gossip message becomes large relative to the data that is transferred. This is shown in Figure 11, in the region where file size is less than 256 KB; the mean download time never goes below 4 seconds. Despite this overhead, swarming will eventually outperform client-server for small files as the arrival rate increases. Nevertheless, this is clear evidence that swarming web servers can benefit from dynamic server initiation.

G. Block Size

As can be seen from our discussion of file size, block size is a key parameter for swarming. To fully explore the effect of block size on swarming performance we conducted a series of simulations with varying block and file sizes, using an arrival rate of 16 clients per minute. Other details of the simulation

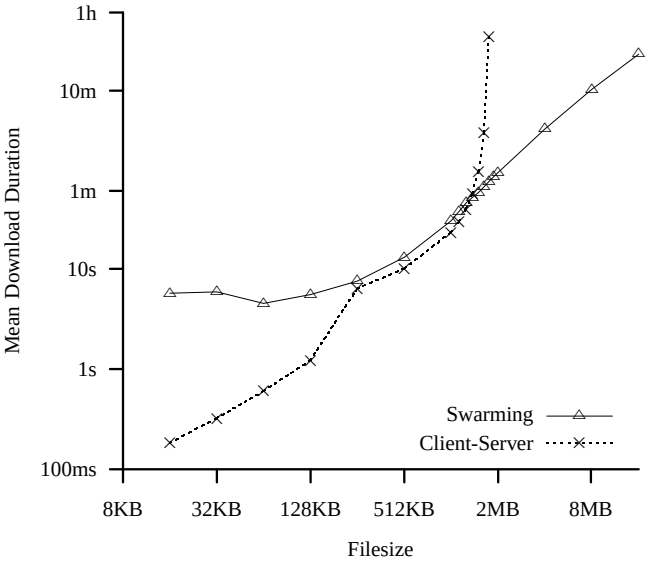


Fig. 11. Impact of file size on performance

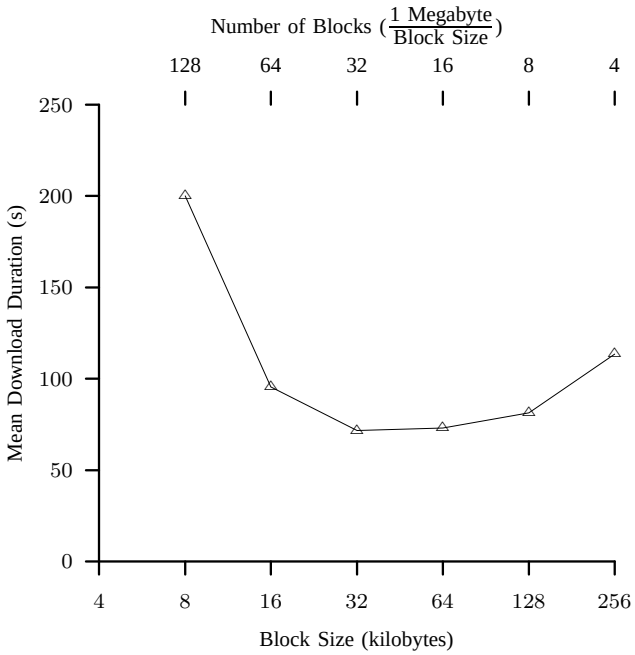


Fig. 12. Effects of block size for a 1 MB file at 16 clients per minute

are the same as in Table II. Figure 12 shows the results of one these simulations, using a file size of 1 megabyte.

From these results we can identify two trends. First, download time increases as the block size decreases. Recall that the client first exchanges gossip messages with the root server or a server peer before requesting a block. The client's upload bandwidth is the bottleneck in this exchange. Hence, as the block size becomes smaller, the transmission delay incurred by the client transmitting a gossip message becomes a significant part of the overall delay.

The second trend in these results is that as the block size increases the download time increases slightly for large blocks. This is a result of the "last block problem", which

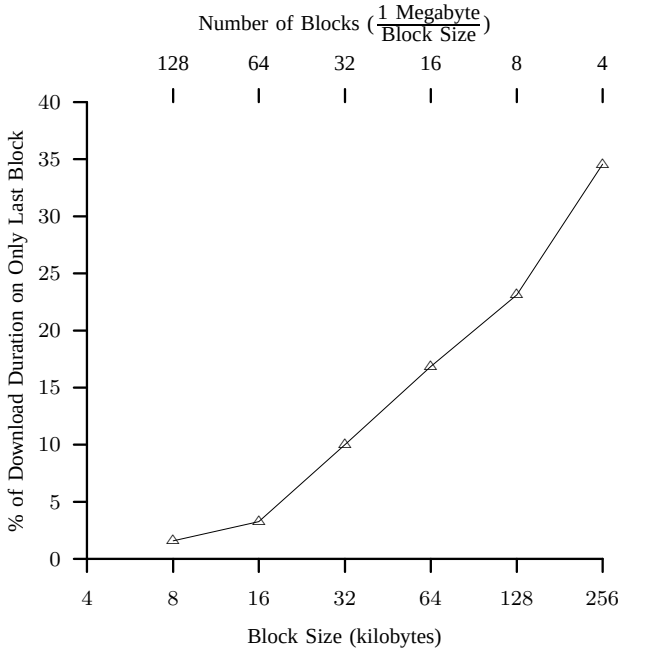


Fig. 13. The last block problem for a 1 MB file at 16 clients per minute

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 most of the file was transferred quickly. Figure 13 illustrates this effect for a 1 MB file transfer, plotting the percentage of time spent transferring only the last block. This graph shows that for a block size of 256 KB the last block consumes 35% of the download time. BitTorrent solves this problem by simultaneously downloading the last block from multiple sources. While this results in redundant data transmission, it can potentially improve download times.

H. Concurrent Downloads

One interesting question for swarming is whether clients are able to improve their performance by increasing the number of concurrent downloads (N_d). We investigate this swarming parameter in Figure 14 by plotting the mean download time as a function of N_d, using two different client arrival rates. The swarming scenario is the basic scenario given in Table II

Figure 14 shows that increasing concurrency does improve performance for higher loads because more clients are active. Just as importantly, increasing N_d does not adversely impact performance for lower loads. Note that the increase in download time due to the increase in load from 8 to 16 clients per minute is consistent with Figure 4.

To explore this issue in more depth, we plot the dynamics of peer selection for this same scenario at 16 clients per minute. As Figure 15 illustrates, clients are able to download from a maximum of about 5 peers at a time, despite raising N_d to 32. Clients do in fact download from (and serve) a greater number of peers as the concurrency limit is increased; however, because the number of active peers is generally about 7 they are unable to find enough active peers with

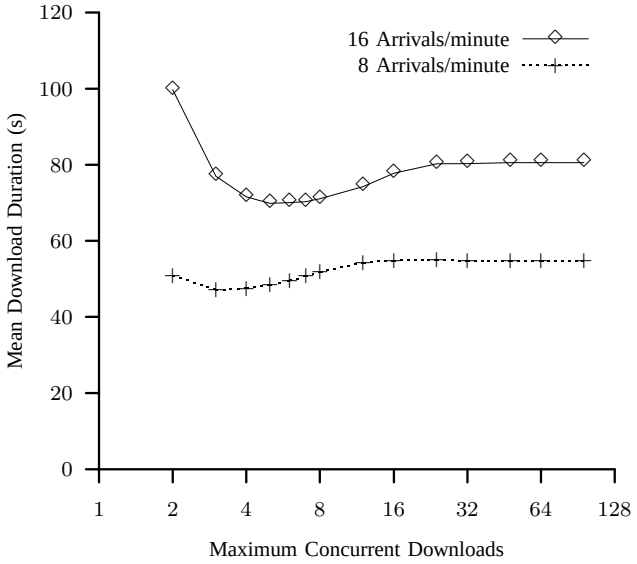


Fig. 14. Impact of concurrent downloads (N_d)

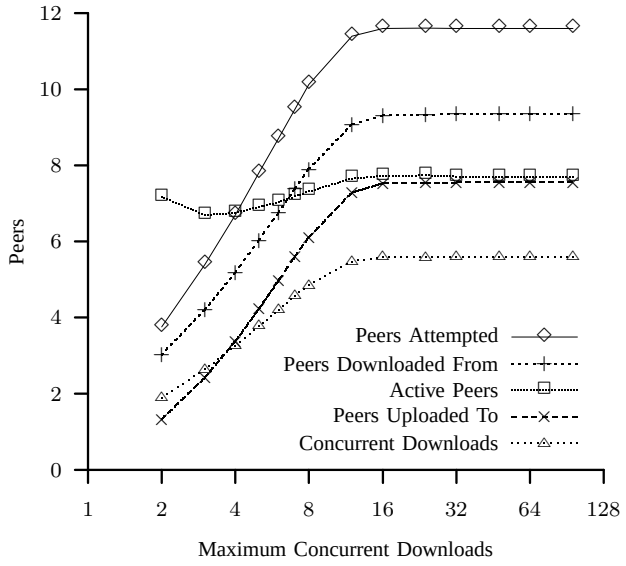


Fig. 15. Peer dynamics for concurrent downloads at 16 clients per minute

their desired content. This indicates that the load is not yet high enough to fully exploit the level of concurrency we are allowing. Furthermore, we should see additional concurrency as we allow peers to have a non-zero lingering time.

I. Client Distribution

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

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

TABLE III
CLASSES OF USERS

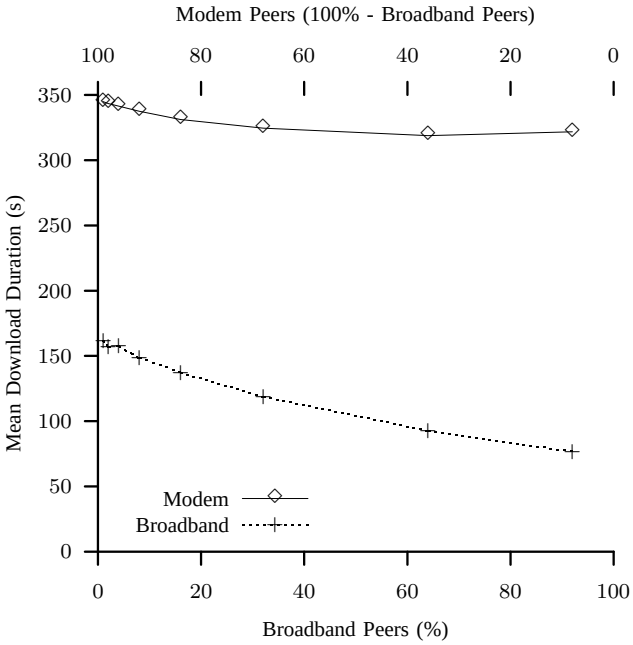


Fig. 16. Impact of low-capacity clients

of each class of users. We assign each class of users a different probability, then randomly assign new clients to one of these classes according to these probabilities. Other than client bandwidth, the rest of the scenario is the same as Table II.

Our study shows that swarming behaves well when low-bandwidth clients interact with higher speed clients. As an example of our results, Figure 16 plots the mean download time for broadband and modem users. In this figure, only broadband and modem users are represented, so as the percentage of broadband users goes down, the percentage of modem users goes up.

From this figure, it is clear 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 increases by roughly a factor of two. While this is a significant increase, the system clearly continues to function well despite an overwhelming number of low-bandwidth users. We see a similar result for office users – their mean download time increases by a factor of 3 for the same changes in the mix of broadband and modem users. The performance of modem users is relatively unchanged by large numbers of higher-speed users because their access link remains a bottleneck.

Our results also demonstrate that broadband users do not see a significant performance increase when small numbers of office users participate in swarming. This is a side-effect

of two aspects of our conservative swarming implementation. First, we are using a lingering time of zero, so that office users do not stay around for long periods helping slower users. Second, clients are not doing any kind of bandwidth-based peer selection. Introducing this latter mechanism should enable clients to take advantage of friendly office users. At the same time, faster users should be able to place a cap on the amount of bandwidth they dedicate to swarming in order to protect both their own performance and the performance of their local network.

V. CONCLUSIONS & FUTURE WORK

Our results show that swarming scales with offered load up to several orders of magnitude beyond what a basic web server can manage. This is an important result, given swarming's popularity in peer-to-peer file-sharing systems. Most impressively, swarming responds quickly to flash crowds, with only a slight increase in download time during the crowd and a rapid return to lower download times once the system returns to steady state. These results confirm that swarming is an excellent choice for the distribution of multimedia content and software updates.

A closer examination of swarming under heavy load indicates that swarming evenly distributes load among the peers and does not cause significant packet loss at the peers. Operating at high load can cause significant packet loss for the root server, but swarming is still able to operate effectively during this time.

We have also examined a number of key swarming parameters. We find that swarming is sensitive to block size, with blocks on the order of 16 to 32 KB providing good performance for larger file sizes. Swarming also performs well across various combinations of client bandwidth. In particular, low-speed users will naturally decrease swarming performance for broadband users but will not introduce significant problems.

From a practical perspective, swarming does have some drawbacks. Because it may potentially use many TCP connections, swarming may steal bandwidth from regular client-server applications. Creating a mechanism for swarming, and other peer-to-peer applications, to share more evenly is an open research problem. In addition, swarming, like many peer-to-peer applications, faces deployment difficulties when users employ Network Address Translation (NAT), because NAT does not allow for incoming connections without special manual configuration. While various mechanisms can work around this difficulty, it becomes more difficult when two peers use NAT.

Finally, our study lays the groundwork for future research in many interesting areas. Of particular concern is reducing congestion at the root server during high load. A server should be able to switch almost completely to redirection during high load, since many peers will have content to server. Likewise, a dynamic server initiation component should be able to decide when to use client-server transfer (for small unpopular files) and when to use swarming (for large or popular files). Other avenues of research include bandwidth-based and distance-based peer selection, dynamic adjustment of the number of concurrent downloads, peer performance monitoring, and more efficient gossiping. We also plan to explore additional scenarios for swarming, such as non-cooperative peers and the effects of some peers lingering for a long time after their download is complete.

REFERENCES

[1] Bit Torrent. http://bitconjurer.org/BitTorrent/.

[2] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest Information Discovery and Access System. *Computer Networks and ISDN Systems*, 28(1–2):119–125 (or 119–126??), 1995.

[3] J. Byers, M. Luby, and M. Mitzenmacher. Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads. In *IEEE INFOCOM*, April 1999.

[4] Russell J. Clark and Mostafa H. Ammar. Providing Scalable Web Services Using Multicast Communication. *Computer Networks and ISDN Systems*, 29(7):841–858, 1997.

[5] F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping and Random Replication to Support Reliable Peer-to-Peer Content Search and Retrieval. Technical Report DCS-TR-494, Department of Computer Science, Rutgers University, 2002.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[7] eDonkey2000. http://www.edonkey2000.com/.

[8] Gnutella. http://rfc-gnutella.sourceforge.net/.

[9] Richard M. Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *IEEE Symposium on Foundations of Computer Science*, pages 565–574, 2000.

[10] K. Kong and D. Ghosal. Mitigating Server-Side Congestion on the Internet Through Pseudo-Serving. *IEEE/ACM Transactions on Networking*, August 1999.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS*, 2000.

[12] Open Content Network. http://www.open-content.net/.

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

[14] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *Proceedings of the NOSSDAV*, 2002.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and s. Shenker. A Scalable Content-Addressable Network. In *ACM SIGCOMM*, August 2001.

[16] P. Rodriguez and E. W. Biersack. Dynamic Parallel-Access to Replicated Content in the Internet. *IEEE/ACM Transactions on Networking*, August 2002.

[17] Squid Web Proxy Cache. http://www.squid-cache.org/.

[18] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, March 2002.

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

[20] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, August 2001.

[21] Swarmcast SourceForge Project. http://sourceforge.net/projects/swarmcast/.