Enhancing Collaborative Content Delivery with Helpers

by

Joseph Hao Tan Wong

B.Sc., University of British Columbia, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

September 2004

© Joseph Hao Tan Wong, 2004
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Joseph Hao Tan Wong
Name of Author (please print) 30/09/2004 Date (dd/mm/yyyy)

Title of Thesis: Enhancing Collaborative Content Delivery with Helpers

Degree: Master of Science Year: 2004

Department of Computer Science
The University of British Columbia
Vancouver, BC Canada
In the client-server model predominant in today's Internet, the load on a server has a significant impact on the level of service perceived by its clients. In particular, given that a server is connected to its clients via an access link of finite capacity, the data transfer rate to each of its clients drops inversely as the number of clients served by the server. This limitation significantly hampers the scalability of such a setup, especially when long-lived connections are involved in the transfer of large files. It is now commonplace for a particular piece of content on the Internet to experience a sudden spike in popularity and hence also in request rate. Such an occurrence, called a "flash crowd" in Internet parlance, is difficult to address via adjusting the provisioning level alone, as it may come and go so quickly for administrative actions to be taken.

To mitigate the problem of the flash crowd, many solutions have been developed. Some of them involve setting up a load-balancing infrastructure, while others rely upon the altruism of clients to provide additional capacity to handle the increased load. In this thesis, we describe a novel hybrid approach, based on the technique of swarming, for scaling up the delivery of large files to many clients. Our design allows machines on the Internet to contribute their bandwidth resources even when they are not interested in the content being disseminated. These peers, known as helpers, are specially tuned to maximize their upload rates while keeping their own download rates to a minimum. We evaluate our approach through simulation, testing it under a variety of conditions involving different bandwidth capacities and node arrival rates. Our results show that helpers are effective in contributing their bandwidth resources under all circumstances, and are able to increase the aggregate upload capacity of the whole system.
Contents

Abstract ii

Contents iii

List of Tables vi

List of Figures vii

Acknowledgements ix

Dedication x

1 Introduction 1

2 Related Work 4
   2.1 Peer-to-Peer Web Caching 4
   2.2 Collaborative Parallel-Download Systems 6
      2.2.1 Multicasting Systems 7
      2.2.2 Swarming Systems 9
      2.2.3 Taxonomy 9
   2.3 Swarming Performance 12

3 Design 14
   3.1 Preliminaries 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Operations</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1 Ordering of Block Requests</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2 Connections between Peers</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Helpers</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1 Design Goals</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Mechanism</td>
<td>21</td>
</tr>
<tr>
<td>4 Evaluation</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Methodology</td>
<td>27</td>
</tr>
<tr>
<td>4.1.1 Simulation Framework</td>
<td>27</td>
</tr>
<tr>
<td>4.1.2 Evaluation Metrics</td>
<td>30</td>
</tr>
<tr>
<td>4.1.3 Baseline Test Cases</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Experiments</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1 Setup</td>
<td>32</td>
</tr>
<tr>
<td>4.2.2 A First Scenario: Instantaneous Flash Crowd with Broadband Peers</td>
<td>33</td>
</tr>
<tr>
<td>4.2.3 Varying Arrival Rates</td>
<td>39</td>
</tr>
<tr>
<td>4.2.4 Varying Helpers' Bandwidth Characteristics</td>
<td>46</td>
</tr>
<tr>
<td>4.2.5 Varying the Number of Helpers</td>
<td>49</td>
</tr>
<tr>
<td>4.2.6 Varying $k_{thres}$</td>
<td>53</td>
</tr>
<tr>
<td>4.2.7 Varying $k_{upload}$</td>
<td>56</td>
</tr>
<tr>
<td>4.2.8 Summary</td>
<td>59</td>
</tr>
<tr>
<td>5 Conclusion and Future Work</td>
<td>61</td>
</tr>
<tr>
<td>5.1 Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>63</td>
</tr>
<tr>
<td>Bibliography</td>
<td>65</td>
</tr>
</tbody>
</table>
Appendix A  Glossary

A.1 Terms ................................................................................. 68
A.2 Notations ............................................................................. 69
List of Tables

2.1 Differences between existing swarming and multicast streaming systems ..................................................... 11
2.2 Taxonomy of collaborative parallel-download systems ............................................................... 12
4.1 Evaluation Metrics ................................................................................................................................. 31
4.2 Default simulation parameters .............................................................................................................. 33
4.3 Simulation parameters for the first scenario .......................................................................................... 34
4.4 Average download times for the first scenario ...................................................................................... 37
4.5 Simulation parameters for varying arrival rates .................................................................................... 39
4.6 Simulation parameters for varying the mix of helpers ......................................................................... 47
4.7 Simulation parameters for the test suite ................................................................................................. 50
4.8 Simulation parameters for varying the number of helpers ................................................................. 50
4.9 Simulation parameters for varying $k_{thres}$ ....................................................................................... 53
4.10 Simulation parameters for varying $k_{upload}$ .................................................................................. 56
List of Figures

4.1 Pseudocode for the maximum-share bandwidth allocation algorithm . 29
4.2 Total number of blocks transferred to regular peers ................. 35
4.3 Useful throughput levels ....................................... 35
4.4 Cut flow levels ...................................................... 36
4.5 Total number of blocks transferred to helpers ....................... 37
4.6 Aggregate helper download rate .................................. 38
4.7 Aggregate helper upload rate ..................................... 38
4.8 Total number of blocks transferred to regular peers ............... 40
4.9 Useful throughput levels .......................................... 40
4.10 Total number of blocks transferred to helpers ...................... 41
4.11 Aggregate download rates and total upload capacity ............. 42
4.12 Time of download completion vs. start time ....................... 42
4.13 Download time vs. start time .................................... 43
4.14 Total number of blocks transferred to regular peers ............ 43
4.15 Useful throughput levels ......................................... 44
4.16 Cut flow levels ..................................................... 45
4.17 Aggregate upload and download rates of the helpers ............. 45
4.18 System throughput results for instantaneous arrival scenarios . 47
4.19 System throughput results for phased arrival scenarios ........ 48
4.20 Helper efficiency results for instantaneous arrival scenarios .. 48
4.21 Helper efficiency results for phased arrival scenarios .......... 49
4.22 Uplink utilization of helpers .................................................. 51
4.23 Downlink utilization of regular peers ................................. 52
4.24 Useful throughput levels for scenarios with helpers on asymmetric links ............................................. 53
4.25 Aggregate helper download rates for scenarios with helpers on asymmetric links ............................. 54
4.26 Useful throughput levels for scenarios with helpers on symmetric links ......................................................... 55
4.27 Aggregate helper download rates for scenarios with helpers on symmetric links ............................................. 55
4.28 Useful throughput levels and aggregate helper download rates for scenarios with helpers on asymmetric links ............................................. 57
4.29 Useful throughput levels and aggregate helper download rates for scenarios with helpers on symmetric links ............................................. 58
Acknowledgements

From the bottom of my heart, I would like to express my sincere gratitude to Dr. Norman Hutchinson, my supervisor and mentor. This thesis could not have come to fruition without the constant support, guidance, patience and inspiration he has shown me through the ups and downs of my research.

I would also like to thank Dr. Charles Krasic, for his insights into the world of content delivery and peer-to-peer systems, and for the valuable feedback he has given me on my research and this thesis.

I am also very proud to have been a part of the DSG, and I would like to thank Dima, Chamath, Geoffrey, Ken and everybody in the DSG for all their advice, tips and tricks, and for cracking me up with laughter every day I come into the lab. The DSG has been a most cheerful and comfortable place to be my home away from home.

Finally, I would like to express my deepest gratitude to my parents for their unwavering love, support and care through it all. I would not have been able to pull it off without them.

JOSEPH WONG

The University of British Columbia
September 2004
To the memory of my loving grandmother.
CHAPTER 1

Introduction

A common occurrence in today's Internet is the phenomenon of the flash crowd, where in a short period of time a server experiences a dramatic spike in its request rate due to a sudden increase in the popularity of the content it hosts. Often, this causes the server in question to grind to a halt, as it no longer has the computational capacity to process the flood of connection attempts. The common solution to such a computational bottleneck is to deploy a load-balanced cluster of servers to handle the incoming requests. In many circumstances, however, a flash crowd's might lies not in its computational requirements, but its burden on the server's bandwidth [19]. This effect is most apparent when the popular content in question is a set of large files, which magnifies the total number of bytes that needs to be delivered. While many servers are connected by links adequately provisioned for regular levels of traffic, the intensity of hundreds of requests per minute can easily cause even well-provisioned links to become congested.

There are several solutions now in common use to mitigate the effects of flash crowds, all of which are different approaches in tackling the same problem, namely that of how to add more bandwidth to the delivery mechanism. Mirroring is the use of alternate servers to host replicas, or mirrors, of the popular content. Clients are then redirected, either automatically, or through a set of links on the
main web site, to these mirror servers to fetch the files. The mirror system possesses, in aggregate, the combined upload capacity of all its constituent nodes in the network. While mirroring is widely adopted and often effective, a mirror server can only help take a share of the load after it has mirrored the content. This means that the copying of content to mirror sites must occur before the flash crowd comes in order to be effective, and hence such a setup must be planned in advance.

Content distribution networks, or CDNs, are another commonly used platform for the delivery of popular content [1, 13, 27]. A CDN consists of a network of nodes scattered across the internet in different geographical locations, each of which serves as a cache of the popular content. The geographic diversity of these nodes means that any particular client could be redirected to a CDN node close by, thus achieving low latency and minimizing network stress. However, as CDNs are usually commercially deployed, a site wishing to utilize a CDN to handle flash crowds must be willing and able to pay a commercial operation for its services.

Whereas both of the preceding approaches operate on a file-by-file basis, delivering data in a some-to-all pattern, cooperative distribution aims to aggregate the bandwidth resources of individual client nodes to aid in the dissemination of files, and operates at a granularity below that of an entire file. Such systems are often touted as scalable, or self-scalable [19, 11], since the aggregate upload capacity of the system increases in some proportion to the demand, represented as the aggregate download capacity of the system. Among the various different flavours of cooperative distribution, swarming is a technique that has caught on in popularity in recent years [9, 26, 30], championed by its most famous implementation BitTorrent. Simply put, in a swarming system such as BitTorrent, a large file is divided into blocks, and clients, or peers, download these blocks from each other in parallel. The key to its efficiency lies in its mechanism for evenly distributing the blocks to different peers, so that any two peers coming into contact will have some blocks that are not in common to share with each other.
In swarming systems, a fundamental assumption is that all peers are interested in the file which they are delivering to one another, as ultimately each peer's main objective is to download the file as fast as it can, while uploading already collected blocks remains only a secondary concern – an entry fee into the system, so to speak. As a consequence, the performance of the system depends heavily on the level of altruism of the participating peers.

In this thesis, we explore an alternate, hybrid approach to the problem of providing additional bandwidth for scaling up a content delivery service. Building upon the foundation of a swarming system, we introduce the notion of a helper, whose purpose is to serve as an on-demand cache in a swarm. The helper would aim to saturate its own upload capacity while keeping its own download rate to a minimum. Our design of the helper is able to provide additional usable bandwidth to the system regardless of the level of altruism of the clients, and for any provisioning level of the helper's link to the network – including symmetric and asymmetric links (e.g., DSL links). This means that helpers can be deployed on almost any node on the Internet and be able to contribute its bandwidth resources under any condition, and as such could serve as the basis of an economical and more adaptive CDN.

The rest of this thesis is organized as follows. In Chapter 2, we discuss previous research in the area of flash-crowd mitigation, and present a taxonomy on existing and potential collaborative parallel-download systems. We present the design of our swarming framework and our helpers mechanism in Chapter 3. In Chapter 4, we give a detailed account of our evaluation methodology and setup, and evaluate our design with a diverse set of experiments covering many operation scenarios. We conclude this thesis and outline some areas of future work in Chapter 5.
Related Work

At the heart of any flash-crowd mitigation technique is a mechanism for replicating content. In order to spread out the high load experienced by a server onto a set of other nodes, these nodes themselves must have either complete or partial replicas of the content with which to serve the redirected requests. Traditional approaches for tackling this problem include mirroring and utilizing commercial content distribution networks (CDNs). Both of these approaches are proactive in nature, as the decision to replicate the content needs to come in advance of any flash crowds. There is always a chance that the flood of requests for a particular piece of content never materializes, and hence the effort and resources spent in replicating the content to all the mirror servers or in contracting out to a CDN is wasted.

2.1 Peer-to-Peer Web Caching

The widely-deployed technique of web caching is another popular approach in addressing flash crowds. Such systems tend to be more reactive with respect to their replication policy, as objects are replicated based on their demand. To achieve scalability, web caching systems employ collections of cooperating nodes acting as caches. Early web caching systems include the hierarchical cache [8] and the summary cache [10].
In recent years, there has been much interest in peer-to-peer systems, and several caching schemes employing peer-to-peer techniques have emerged. They offer the benefit that every client can potentially act as a cache as well, thus allowing the system to scale with the number of clients.

A few of these systems rely on the construction of structured overlay networks in the form of distributed hash tables (DHTs). Squirrel [14] is a decentralized, cooperative web cache based on the Pastry [24] overlay network, while Backslash [28] is a similar system currently implemented on top of the CAN [23] overlay. In Backslash, nodes participating in the system are themselves web servers which would like to be protected from flash crowds. As such, a Backslash node would function as a web server in normal operations, and would only redirect onto the overlay network when it becomes overloaded by high request rates. In contrast, Squirrel involves all clients in its overlay, and routes all requests through the overlay. Nonetheless, these two caching overlays operate on almost identical principles.

When an HTTP request is routed through the overlay in either Backslash or Squirrel, the URL is hashed into a key in the overlay’s ID space. In their first mode of operation (called home-store in Squirrel and local diffusion in Backslash), an object with URL \( i \) is cached on the node whose nodeID is closest to SHA-1(\( i \)) (called the home node for \( i \)). When the request rate hits a predetermined threshold, the object is replicated onto the previous node on a requestor’s route to the home node. By doing so, a node that locally observes a flood of requests creates a bubble of caches around itself, thus diffusing the load. In their directory mode, both systems maintain on \( i \)’s home node a directory of pointers to nodes which have recently requested copies of the object, some of which may still have it available to be delivered to other requestors.

PROOFS [29] is another peer-to-peer caching scheme. It differs from Backslash and Squirrel in that it utilizes a randomized overlay of client and server nodes for caching web content. Clients send out queries for objects on the overlay net-
work, and at the same time can opt to cooperate by answering queries from other nodes. Compared to the approaches using structured overlays, PROOFS has to solve the additional problem of locating replicas in a network whose topology is random and constantly changing.

While these systems have been demonstrated to be effective for handling high request rates, they do so without consideration for bandwidth issues. For example, if a large file (e.g., a Linux distribution) is being popularly requested, none of these systems would attempt to place replicas of this file on well-endowed nodes with high upload capacities, which would be required to sustain long-lived upload connections with the many requestors. Hence, these systems are not especially suited to the job of mitigating high demands for large files.

### 2.2 Collaborative Parallel-Download Systems

Many systems have been devised especially to handle the delivery of large files. They overcome the difficulties of replicating large files by breaking up a file into smaller units of replication and transfer called blocks. Instead of locating and downloading a replica of a file among many files stored in a web caching system, a peer in one of these systems would instead be locating and downloading a replica of a block among the many blocks of a file, and would ultimately want to download every block of the file. The issue of bootstrapping onto the right overlay for a particular file is now a separate problem.

Since a file is now broken into many blocks, and each peer would want to download every single block, these systems take advantage of the opportunity to increase the throughput by letting peers download multiple blocks from multiple peers simultaneously. These collaborative parallel-download systems can be grouped into two general categories: multicasting systems and swarming systems.
2.2.1 Multicasting Systems

As shown in [4, 3, 17], it is feasible to deliver a file to a large number of clients using multicast when the file is encoded by an erasure encoding. Castro et al. advocated in [5] the use of an application-level multicast system to deliver bulk data. In order to fully utilize the upload capacity of participating peers, their system, called SplitStream, organizes the peers into a forest of trees which share no interior nodes with high probability.

SplitStream is built upon the Scribe multicast system [7, 6, 25], which in turn is based upon the Pastry DHT infrastructure. Each SplitStream node therefore joins the system with a Pastry nodeID. The file to be delivered is encoded as $k = 2^b$ distinct streams called stripes. Each stripe is identified by a stripeID, which has the property that the stripeIDs of all stripes differ in the $b$ most significant bits. These stripeIDs serve as multicast group names in the Scribe layer. A peer subscribes to a multicast group for a stripe by routing a message to the stripe's root, which is the peer whose nodeID is closest to the stripeID. An application-level multicast tree is formed using reverse-path forwarding from the root through the overlay, and each stripe is streamed from the source through its own stripe tree. Pastry's prefix routing mechanism entails that in a SplitStream system where all the trees are built this way, a peer will be an interior node for at most one stripe tree, namely the one whose stripeID shares the $b$ most significant bits with the peer's nodeID. Through this mechanism, the majority of peers in SplitStream participate as interior nodes, and so they each help to take a share of the load away from the original data source.

This tree-building mechanism resorts to a fallback plan, however, when a peer is not able to join a stripe tree because of bandwidth constraints. SplitStream allows each peer to specify its bandwidth constraints in terms of the number of stripes it is able to receive and forward. This is to guarantee that every interior node of a stripe tree has the capacity to forward that stripe. When a stripe tree runs out of upload capacity, represented by free children slots on interior nodes,
SplitStream concedes by asking some other peer with excess upload capacity to join the stripe tree as an interior node. This other peer may in fact be an interior node for some other stripe tree. In such a case, the trees are no longer interior-node-disjoint. However, this does not affect the link utilization of the system, only its resilience to node failures.

To enable parallel downloads, a peer joins as many stripe trees as it can. With the redundancy afforded by an appropriate erasure encoding scheme, the peer will be able to reconstruct the file even when it has not been receiving all stripes, or has missed some blocks within a stripe.

Bullet [16] is another streaming system explicitly supporting the transfer of large files. Recognizing that the bandwidth of a tree is guaranteed to be monotonically decreasing moving down the tree, every interior node in Bullet's only tree, called the backbone tree, may choose not to forward the entire stream it receives to every one of its children. Instead, the node can drop blocks when its outbound connections are congested. Bullet's algorithm is configured such that in these scenarios, a node will forward sets of blocks which are maximally disjoint to its children. This maximizes the ability of siblings to use up any excess upload capacity they have by sharing blocks with each other through connections that exist outside of the backbone tree. A tree-based gossip mechanism allows peers to find out about each other, so as to establish parallel download connections for the sharing of blocks.

Other multicast streaming systems, while not designed specifically for the delivery of files, can similarly be used for this purpose through the use of an appropriate erasure encoding. The streaming protocol of CoopNet [19, 20] is one such system. As with SplitStream, it uses a forest of interior-node-disjoint trees for the delivery of multiple stripes to peers. However, this protocol differs from SplitStream in that there is no lower-level multicasting or DHT substrate. Instead, the responsibility of maintaining the forest topology lies with a central coordinator,
which dictates to which tree a peer should join as an interior node.

2.2.2 Swarming Systems

BitTorrent [9] is perhaps the most well known of the swarming systems. They differ from the multicasting systems in that no explicit dissemination trees are formed. Instead, peers form an ad-hoc mesh through which they transfer data to one another. BitTorrent is popularly used to distribute large files such as software updates and releases. A file is split into many smaller blocks in BitTorrent, and peers in the system download blocks from one another, while at the same time uploading blocks they already have to others. BitTorrent motivates peers to upload by employing a choking technique, where a peer will preferentially upload to those that upload to itself the fastest. Hence peers that do not upload much also tend to receive worse treatment from other peers connected to them. All peers interested in a particular file register with the central coordinator responsible for the file called a tracker, which keeps track of the membership of nodes in the system.

Slurpie [26] is another cooperative protocol for bulk data transfer based on swarming. It differs from BitTorrent in that the technique of gossip is used in place of the centralized tracker to inform peers in the system of the arrival or departure of other peers. It also attempts to achieve better performance by dynamically adjusting the number of connections based on the bandwidth conditions at the time.

2.2.3 Taxonomy

Swarming and multicast streaming are two current techniques which have the potential to satisfy the needs of file distribution and bandwidth sharing. We have developed a taxonomy for these collaborative parallel-download systems, allowing us to focus upon the salient features which distinguish each system from the others.

In SplitStream and CoopNet, the content to be delivered is divided into one
or more stripes. Each of these stripes is delivered at a fixed rate, and peers subscribe
to as many stripes as they can. The peers which subscribe to a particular stripe form
a dissemination tree, and the stripe’s content then flows from its source down the
tree, at the specified rate. Within a stripe, the content is subdivided into blocks
or data packets, and these are delivered in sequential order. Depending on the
underlying transport, the delivery of blocks may be lossy, and erasure encoding
may be employed to compensate. This same technique is used to handle node
churn, where a node leaving the system causes otherwise non-lossy connections to
become permanently lossy.

In swarming systems, the pertinent content to be delivered is most often a
file, which is subdivided into blocks. Peers form connections with each other in
an ad-hoc fashion and deliver blocks to each other. Intrinsically, a peer connection
does not have a specified delivery rate associated with it, nor does it place a re-
striction on which blocks must or must not be delivered through it. Moreover, to
maximize the utilization of bandwidth, blocks can be requested and delivered in
any order desired, and peers do not need to have the content completely down-
loaded before it can upload blocks to others. Swarming deals with churn naturally
since it is a block-based technique and there is no enforced ordering of the delivery
of blocks.

Bullet sits somewhere in between the two, as its tree connections are not
required to deliver data at any particular rate. In addition to having a tree, it also
permits peers to form ad-hoc connections for sharing blocks, in much the same
way as swarming systems.

We summarize the differences in characteristics between these systems in
Table 2.1.

Among different streaming and swarming systems, we also find other dis-
tinguishing design choices which have a major impact on their modes of operation.
Table 2.1: Differences between existing swarming and multicast streaming systems

<table>
<thead>
<tr>
<th></th>
<th>SplitStream/CoopNet</th>
<th>Swarming</th>
<th>Bullet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
<td>trees</td>
<td>ad-hoc mesh</td>
<td>tree and ad-hoc mesh</td>
</tr>
<tr>
<td>Delivery rate per connection</td>
<td>fixed</td>
<td>variable</td>
<td>variable</td>
</tr>
<tr>
<td>Blocks sent on a connection</td>
<td>only those belonging to the stripe</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>Order of delivery</td>
<td>sequential order</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>Handling of missing blocks</td>
<td>erasure encoding</td>
<td>reconciliatory transfers</td>
<td>reconciliatory transfers</td>
</tr>
</tbody>
</table>

Group management

The management of the group of peers in both streaming and swarming could be centralized or decentralized. In CoopNet, there is a central coordinator that maintains the topology of the multiple stripe trees, whereas in SplitStream, a decentralized DHT is used for the same purpose. In BitTorrent, a tracker serves as a central repository on group membership information, while in Slurpie, a gossip mechanism is used to disseminate group membership information.

Discovery of content

The search for other peers to connect to may be guided by a peer’s desire for particular portions of the content. In essence, this is can be regarded as an extension of the group management mechanism, where group memberships are maintained not at the file granularity but at a lower granularity, e.g., a stripe or a block. A peer would query this mechanism in order to discover other peers with the content it is interested in downloading. In SplitStream and CoopNet, group membership is maintained at the stripe level. With Slurpie, the gossip messages passed between peers contain not only a list of other peers, but also which blocks those peers have.
Together with the characteristics distinguishing streaming from swarming, we have just described seven design decisions that are fundamental to the operation of the content distribution system (see Table 2.2). These choices are in fact orthogonal to one another: they could be mixed and matched to form a whole spectrum of content distribution systems.

### 2.3 Swarming Performance

To date, two groups have performed measurement studies on the performance of swarming, in particular of the BitTorrent protocol [15, 21]. Izal et al. [15] followed the lifetime of a torrent for the 1.77GB Redhat 9 Linux distribution for five months, and analyzed the statistics of the thousands of peers involved. They conclude that BitTorrent is highly effective, and is able achieve throughput levels that are capable of sustaining large flash crowds. In [21], Pouwelse et al. present detailed measurements of BitTorrent taken over an eight-month period, including results on the popularity and availability of files and trackers, the download performance, the lifetime of torrents, and the fraction of corrupted or wrong content.

The performance of swarming has also been studied analytically. Biersack et al. [2] provide several analytic models which approximate the operations of both streaming and swarming systems. They offer some insight into the choice of system parameters such as bounds on node degrees and the number of blocks.
a file should be split into. Through the use of linear programs, Mundinger and Weber \cite{mundinger2006analytical} analytically show how to solve the problem of minimizing the time it takes for a source to distribute a file to \( n \) other nodes. Both \cite{kuhn1955hungarian} and \cite{mundinger2006analytical} assume that the upload capacity is uniform across all peers. Qiu and Srikant \cite{qiu2007modeling} developed deterministic and stochastic fluid models specifically to study the performance of BitTorrent. Through these models, they considered the scalability, performance and efficiency of BitTorrent, as well as the effectiveness of its peer selection and choking mechanism in incentivizing the system.
CHAPTER 3

Design

Our survey of existing collaborative content delivery systems reveals that both streaming and swarming are good candidates upon which to build our augmented delivery framework. For our work, we have chosen to utilize the swarming model as the foundational technology, since it has proven itself as an effective method in the collaborative distribution of large files. In this chapter, we present the design of an extension to swarming that allows peers to volunteer their upload capacity to the benefit of others.

3.1 Preliminaries

We begin our exposition on the details of swarming by outlining the entities involved and explaining some of the terms and notations that we use. All of the terms and notations introduced here and in subsequent sections are summarized in the Glossary (Appendix A) for easy reference. In the interest of facilitating communication, we adopt the terminology of BitTorrent wherever appropriate.

In order to facilitate the use of parallel download connections, a file is split up into many smaller, equal-length fragments called blocks. Each block within a file is identified by a block id. For every peer $A$, let $B_A$ denote the set of blocks that $A$ has.
These file blocks are transferred on connections that are semantically unidirectional, even though the underlying transport, TCP, supports bidirectional connections. As such, we refer to the sending and receiving ends of a connection as the source and the sink respectively.

Each peer is connected by an access link to the rest of the network, and we denote by $UC_A$ and $DC_A$ respectively the upload and download capacity of the peer $A$'s access link in bits per second (bps).

We refer to a set of peers which are uploading and downloading the same file via swarming as a swarm. Coordinating the membership of the swarm is the tracker, a server process responsible for handling peers joining and leaving the swarm. The tracker itself could be hosted by any node, regardless of whether it is in the swarm or not. As the scalability of a centralized coordinator is not our primary concern, we contend that the use of such a tracker is sufficient for our evaluation needs.

Within the swarm, some of the peers may possess all the blocks for the file. We refer to these peers as seeds. Since seeds do not need to download any blocks, their presence in the swarm is simply to serve blocks to others. Setting the seeds aside, the remaining peers in the swarm have only partial copies of the file, and are known as leechers. When a leecher has completely downloaded the file, it changes status and becomes a seed.

Looking at the bigger picture, besides the peers that are actively engaged in swarms, there may be other peers that express no interest in the content distributed by the swarms, but are nonetheless willing to contribute their bandwidth to the aid of others. We call these peers candidates. If a candidate then joins a swarm to offer its bandwidth resources, the peer then becomes a helper in that swarm. In addition, the term regular peer is used to refer to a non-helper peer in the swarm. We note that a helper, just like a regular peer, can be classified as a seed or a leecher depending on whether it has the entire file or not.
3.2 Operations

We now describe the basic operation of our swarming protocol. First of all, it should be noted that this protocol is our own design, and differs from existing swarming protocols like BitTorrent and Slurpie in several aspects. Throughout our explanation, we will point out the various design decisions involved and the choices that we made.

To join the swarm for a file, a peer first contacts the tracker and registers with it. The tracker would add the newly joined peer’s identity into its records. If the peer joins as a leecher, it would then request a list of peers from the tracker. In our design, the tracker returns a list that is uniformly sampled from its records for all peers in the swarm. In particular, a new peer in the system is as likely to be picked as one that has been in the swarm for a while. This makes sure that old peers in the swarm continue to be effectively utilized. Also, the tracker makes no attempt to select peers that have the blocks that the requesting peer is interested in, since maintaining the additional bookkeeping information about which peers have which blocks on a continuing basis necessitates a significant amount of additional communication overhead.

Having received the list of other peers, the leecher then contacts each one in turn to attempt to establish a unidirectional download connection. The connection request can be accepted or rejected by the other peer depending on several factors, which we describe in Section 3.2.2. If a connection is established, the requestor acts as a sink and starts requesting for blocks to be transferred. A short queue, two blocks in length, is set up on the source to make sure that there is always something to send when the connection is ready. The ordering for the block requests follows the rarest-first policy, explained in Section 3.2.1.

When it reaches the point where the sink has all the blocks that the source has, no blocks would be transferred and the connection will go idle. An idle connection will start transferring again when the source receives a new block that is
also desired by the sink. If a connection remains idle after a timeout period, however, the connection will be dropped. The sink then contacts the tracker again to request additional peers to contact if the number of download connections falls below a lower bound.

This entire process continues for a leecher until it has completely downloaded the file, block by block, and becomes a seed, at which point its remaining in the swarm is completely optional. A peer can also leave the swarm at any point in time voluntarily, whereupon it would deregister itself with the tracker. Connections would be broken as the peer departs, leaving partially transferred blocks on its sinks that would then be discarded.

The main effect of an abrupt node failure in a swarming system such as ours is that stale information would be distributed by the tracker. Since it would be easy for a peer to notify the tracker that the list of peers it received contains stale data, we choose to model all peer departures as proper voluntary departures in our evaluation.

3.2.1 Ordering of Block Requests

When selecting which block to download next, a peer would request a block that the fewest of its sinks already have. Once the block is completely downloaded, the peer can then turn around and upload this block to those sinks which do not have this block. This approach is often called the rarest-first policy, since a peer chooses to download a block that it deems is the rarest among its sinks. We employ this approach as it maximizes a peer's potential of having something useful to upload by preferentially downloading blocks that are most wanted by its sinks.

To implement rarest-first, a peer categorizes all block ids by their local availability measure. The local availability of a block $i$ on a peer is defined to be the number of the peer's sinks which have block $i$. Therefore, the measure of local availability for a block ranges from 0 up to the total number of sinks. For each
possible value of local availability, we maintain a randomly permuted list of the
associated block ids. When it is time to choose the next block to download from a
source, a peer would scan these lists one by one, in availability order from lowest
to highest, until it finds a block which is offered by the source but has not been
downloaded.

These lists of block ids are updated dynamically. Recording a change in
availability of a block involves removing the id from its old list and inserting it into
a random position in the new one. Every time a sink connects with or disconnects
from one of its sources, the sink sends a record of the blocks it has to the peer, and
the availability levels on the peer are duly updated. In addition, a sink sends a
status update to all its sources whenever it receives a new block. By doing so, each
peer in the swarm has the most up-to-date record of local availability levels at any
moment in time.

3.2.2 Connections between Peers

The connections in our base swarming system are semantically unidirectional, that
is to say that a peer $A$ downloading from another peer $B$ is not duty-bound to re-
ciprocate and upload to peer $B$ via the same connection. Peer $A$ could in actuality
be uploading to peer $B$, but this connection would be maintained separately. This
represents a departure in design from existing swarming systems such as BitTor-
rent, where there is an emphasis on a per-connection notion of fairness. We choose
to look at fairness in a broader sense, since helpers will by definition be uninter-
ested in the content which they help to deliver, and there is no incentive for helpers
to help at all if their only reward is blocks of a file they are not interested in.

One of the advantages of having unidirectional connections is that the num-
ber of upload connections is decoupled from the number of download connections.
For peers on asymmetric access links with more download than upload bandwidth
capacity, they can have more download connections than upload connections, ren-
dering it easier to saturate both directions of their access links. Another benefit of the decoupling is the added flexibility for a peer to be able to choose sources and sinks independently, according to different criteria.

We place an upper limit on the number of incoming and outgoing connections, since having too many active TCP connections would result in some of them getting starved due to the ad-hoc nature of bandwidth allocation in real TCP implementations. Moreover, having more connections splitting the same amount of bandwidth on a peer's access link would also increase the time required to deliver the first block on each connection. This would have the consequence of slowing down the propagation of blocks through the swarm during the ramp-up period.

On a particular peer $A$, we set an upper limit $N_A^d$ for the number of incoming (download) connections, and an upper limit $N_A^u$ for the number of outgoing (upload) connections. Each limit is lower-bounded above zero so as to allow each peer to maintain a minimum level of parallelism in its download/upload efforts. At the same time, the limit grows linearly as the download/upload bandwidth capacity of the peer, so that each additional connection represents a fixed amount of additional capacity. By doing so, we bound the heterogeneity of the capacities of the various connections within the swarm, thereby rendering it less important for a peer to evaluate existing or potential connections based on transfer rates.

This approach is similar to what is done in SplitStream and CoopNet, where the number of outbound connections on a peer is simply the outbound bandwidth capacity divided by the bandwidth requirement of one stripe. Where we differ from these two systems is that the actual attained transfer rate on each connection is not rigidly enforced, but is rather attenuated by the underlying TCP flow control mechanism.

With the upper limit on the number of outgoing connections in place, a peer $A$ needs to make a decision whenever a connection request arrives and it has already reached this upper limit. The peer has the choice to either reject this request,
or to evict one of its existing connections. As its first step, the peer checks to see that it has some blocks to offer which the potential sink does not have. If it can find no such block, the request is rejected. Otherwise, the peer $A$ computes for each peer $p \in Sinks \cup \{requestor\}$ the set difference $\delta_p = B_A - B_p$ and a score $s_p = |\delta_p|$. Here, the score $s_p$ represents the number of blocks peer $A$ could currently send to peer $p$. Since the main goal is to try and maximize the opportunity to upload blocks, the peer chooses to dissociate with peer $p_{\text{victim}}$ whose score $s_{p_{\text{victim}}}$ is the smallest among all $p \in Sinks \cup \{requestor\}$. If $p_{\text{victim}}$ is the requestor, the connection request is rejected. If not, the request is accepted, and the existing connection to $p_{\text{victim}}$ will be broken after the current block in transit is completely transferred.

To foster stability in the topology, we slightly bias the system to prefer existing connections by additionally imposing a small penalty on the requestor. In particular, we set $s_{\text{requestor}} = k_{\text{penalty}} \cdot |\delta_{\text{requestor}}|$, where $k_{\text{penalty}}$ is a scaling factor $\leq 1$.

3.3 Helpers

3.3.1 Design Goals

By itself, swarming enables nodes interested in the same file to cooperatively upload parts of the file to each other. From the perspective of the original content provider, its clients are collaborating to take a large share of the load away from itself. In extending the swarming model, we aim to satisfy two related goals: 1) the mechanism should enable a content provider to invite additional nodes into the delivery network with the sole purpose of providing additional bandwidth, and 2) the mechanism should enable nodes which are otherwise uninterested in the content to volunteer their bandwidth resources and make a positive contribution to the delivery.

With the first goal, our motivation is to allow a content provider to have
control in increasing the capacity of its delivery network on an on-demand basis. In turn, this enables a variety of cooperation arrangements. For example, a group of content providers may form a mutual cooperation pact, where the nodes of each provider are willing to be called into action in carrying traffic for another provider swamped by a flash crowd. In another scenario, there may be a collection of nodes that are willing to handle redirected traffic from any content provider that asks, effectively forming a public service similar to that of a CDN.

Our second goal focuses upon the ability of an individual node to make a contribution without actually wanting the file it is distributing. In many peer-to-peer file distribution systems, there is often an emphasis on a per-connection notion of fairness, where the speed at which a node is able to download is correlated to the amount of upload bandwidth it gives back to the system. For example, a node connected to the Internet via an asymmetric broadband link may not be able to download at full speed because its upload bandwidth capacity is only a fraction of that amount. We propose to alleviate this problem by allowing nodes to accrue some form of payment by transferring content which they otherwise do not want, so that they will be able to pay for better performance later on when downloading the content which they do want. Our mechanism shall be able to serve as the technical underpinnings of such an endeavour; however, the design of the complementary economic or incentive system is beyond the scope of this thesis, and represents one of the directions of our future research.

3.3.2 Mechanism

For the design of our mechanism, we draw some insights from the streaming model. Let us consider a streaming system whose content is divided into $m \geq 1$ stripes of equal bandwidth, and each stripe is delivered in its own dissemination tree. By looking at how many stripes a peer is receiving and how many children it has, it is straightforward to gauge whether the peer is a net consumer or provider of ca-
pacity to the system. In particular, if a peer is receiving fewer stripes than it has children, it is giving more than it consumes, and is thus "helpful" to the system. Therefore, it is easy to see that a peer that is strictly interested only in helping could do so by signing on to as few stripes as possible while maintaining a full complement of children. Every block received by such a helpful peer is forwarded to one or more receivers, and the peer always downloads blocks at a rate no greater than its total upload rate.

In the swarming world, there is no notion of stripes, nor fixed-bandwidth connections. A connection that is usefully transferring data between a source and a sink at one moment may go idle at the next when the source no longer has any blocks that the sink wants. Therefore, it is not enough to simply look at the number of incoming and outgoing connections of a peer to judge whether it is a net consumer or provider of bandwidth capacity to the system. Matters are much more dynamic in a swarming system.

From the observation that a helpful streaming peer downloads slower than it uploads, we employ an adaptive rate-limiting mechanism to achieve the same effect in a swarming peer. It operates on the same simple premise: provide as much upload capacity as possible while consuming no more download capacity than it must.

With streaming, one of the properties of a helpful peer $A$ is that every block it receives is forwarded to at least one other peer, or more generally, $uf_A$ other peers, where $uf_A \geq 1$ is called the upload factor for $A$. For our design of a helper peer in a swarming system, we would want to simply enforce this property as a policy. However, since a block must be downloaded before it could be uploaded, a helper cannot guarantee that a block it is about to download will, in the future, be uploaded at least $uf_A$ times. We have to settle with an approximation. While a helper cannot guarantee that a block will be uploaded in the future, it can at least confirm that $uf_A$ or more of its sinks are currently missing this block and would
therefore have the desire to download it when it becomes available. This check plays a crucial role in our rate-limiting mechanism as it prevents a helper from downloading blocks whose demand is low or non-existent among its sinks.

Just as importantly, a helper needs to control its downloading of blocks so that, on average, it consumes less than it provides. Here, we continue with the insight that each block downloaded by a helper $A$ should be uploaded at least $u_f A$ times. At any point in time, a helper may have some blocks which it has downloaded but has each been uploaded less than $u_f A$ times. We refer to these blocks as unfulfilled blocks as their purpose of being uploaded at least $u_f A$ times has yet to be fulfilled. For every block $i$, the helper maintains the number of times it has been uploaded, denoted $u_A(i)$. Therefore, the set of unfulfilled blocks for a helper $A$ is represented by $\Phi_A = \{i | u_A(i) < u_f A\}$, and we define $\phi_A = |\Phi_A|$ to be the number of such unfulfilled blocks.

If the helper is able to download faster than it can upload, and it does not stop downloading, the number of unfulfilled blocks will grow. Therefore, a natural way to limit the rate of download is for the helper $A$ to refrain from requesting blocks while the number of unfulfilled blocks, $\phi_A$, is above a certain threshold $T_A$. Doing so gives the helper some time to upload the unfulfilled blocks without having new blocks added to the unfulfilled ranks.

This arrangement has certain salient characteristics. First, as a helper $A$ starts off with no blocks, it behaves like a regular peer for at least the first $T_A$ blocks it downloads. This allows the helper to quickly obtain a few blocks which it could upload to other peers. Also, an increase of 1 to $\phi_A$ requires the uploading of $u_f A$ blocks worth of data to offset. In effect, this limits the download rate of the helper to at most $1/u_f A$ of its upload rate, averaged throughout the lifetime of the helper.

For a helper peer $A$, we set

$$u_f A = k_{upload} \cdot N_A^u,$$  \hspace{1cm} (3.1)
and

\[ T_A = k_{\text{thres}} \cdot UC_A, \]  

(3.2)

where \( k_{\text{upload}} \) and \( k_{\text{thres}} \) are system-wide parameters. The setting for the upload factor can be interpreted as a wish for a helper to upload each block to a certain fraction of its sinks, however many or few there are. At the same time, we choose to set the threshold on a per-helper basis because a helper with a faster uplink would want to have more blocks to offer its sinks early on so as to keep the pipelines full and the uplink fully utilized.

Once a helper \( A \) downloads a particular block \( i \), that block remains in the unfulfilled state until it has been uploaded \( u_f_A \) times. However, it is possible that since the time when the helper first requested block \( i \), the situation has changed. Let us denote by \( \text{desire}_A(i) \) the number of \( A \)'s sinks which do not have the block \( i \). While \( \text{desire}_A(i) \geq u_f_A \) when the decision was made to download block \( i \), it may now be the case that \( \text{desire}_A(i) < u_f_A \). If so, we reevaluate the block, and artificially modify the upload count of the block, \( u_A(i) \), as follows:

\[ u_A(i) \leftarrow \max(u_A(i), u_f_A - \text{desire}_A(i)) \]  

(3.3)

This then allows the block \( i \) to be taken off the unfulfilled list after it has been uploaded another \( \text{desire}_A(i) \) times, at which point all the current sinks will have downloaded block \( i \). The helper performs this reevaluation periodically with period \( \tau_{\text{reeval}} \) whenever \( \phi_A > T_A \). By performing this reevaluation, the helper makes sure that no blocks are stuck in the unfulfilled state indefinitely, which if left untreated would obstruct the helper from downloading further blocks.

A side benefit of the slow download rate of helpers is that over the course of the helpers' participation in the swarm, they tend to consume less local storage for the caching of blocks. This makes it possible for helpers to aid in the delivery of more swarms simultaneously.
Inviting Helpers

In order to be able to determine whether the aid of helpers is necessary in the delivery of a particular file, we need to know whether the demands of the regular peers are satisfied. At any point in time, given the full knowledge of the peers in the swarm and their link capacities, we can calculate the aggregate upload and download capacities of the swarm. We observe that in an ideal situation, where there is more aggregate upload capacity in the network than there is aggregate download capacity, there is no room for further improvements, as all the peers are downloading as fast as they could. Therefore, we use the demand/supply ratio \( D/S \), calculated as

\[
D/S = \frac{\sum_{i \in \text{regular peers}} DC_i}{\sum_{i \in \text{all peers}} UC_i}
\]  

(3.4)
to determine our utilization of helpers.

The use of the \( D/S \) ratio requires the system to keep additional bookkeeping information. In particular, the tracker maintains the download and upload capacities of every peer in the swarm in terms of bits per second. We do not require these estimates to be extremely accurate; information statically supplied by the users from their knowledge of their access links (cable, ADSL, T-3, etc.) is sufficient. In our accounting scheme, we count only the download capacity of regular peers towards the aggregate download bandwidth. Helpers are excluded from the tally because through the use of our rate-limiting mechanism, helpers should exert very little demand on bandwidth.

When the tracker notices, during the course of its operations, that the \( D/S \) ratio exceeds an upper threshold \( r_{\text{invite}} \), it invites candidates into the swarm as helpers to satisfy the excess demand. These candidates register themselves with a central, well-known candidates coordinator. It is from this coordinator that the tracker finds out about the identities of these candidates. Once it has obtained such a list of peers, the tracker contacts the candidates one by one and requests that they join in the delivery of the file as a helper. The invitation process stops when
the $D/S$ ratio is restored to be below $r_{\text{invite}}$, or when the number of helpers in the system hits an upper limit. We limit the number of helpers to be no more than $k_{\text{helper}}$ times the number of regular peers in the swarm, so as to ensure a minimum representation by regular peers.

Conversely, when the $D/S$ ratio falls below a lower threshold $r_{\text{uninvite}}$, the tracker can request some of the helpers already participating in the file delivery to leave the swarm. While this may have the potential of reducing the download performance perceived by regular peers, this mechanism helps to conserve the utilization of helpers, especially at times when they are not overwhelmingly needed. The end result is that the helpers can now offer their service to other swarms in need.
We have designed and conducted a set of experiments to evaluate the effectiveness of our helpers mechanism. These experiments were conducted in a simulation environment, and under a variety of parameter settings and workloads. The results obtained from these experiments support our hypothesis that helper peers of our design do indeed make a beneficial contribution to the performance of swarming systems.

4.1 Methodology

4.1.1 Simulation Framework

In order to support the simulation of a large number of nodes and links under high loads, we developed a custom network simulator based on flow-level modelling [12], and implemented it in 1500 lines of C++. For our simulations, we use a simple star topology, with each peer node connected to a central hub via its access link. Our choice of the star topology is based on the assumption that, in the Internet, the bottlenecks of swarming are at the access links of the peers, and not within the core. We refer to the outgoing and incoming portions of a node's access link as its uplink and downlink.
The fundamental construct of our simulator is the notion of a flow. Semantically, a flow is a unidirectional data connection between a source and a destination, and corresponds to an approximation of one half of a bidirectional TCP connection. Packets are transferred from the source to the destination through the flow, and each flow is equipped with a send queue for packets waiting to be sent. A flow that has no packets in transit is said to be idle.

At any particular moment in time, a non-idle flow is transferring data at a flow rate equal to its \textit{bandwidth allocation}. The bandwidth allocation for a flow can change with time, and can even change multiple times during the transfer of a single packet.

In existing flow-level simulators, the technique of \textit{minimum-share allocation} is often used to allocate bandwidth to flows. Using this allocation scheme, a node’s uplink bandwidth capacity is split into equal bandwidth shares among all the flows that originate at the node. Similarly, a node’s downlink bandwidth capacity is split among all flows terminating at the node. The bandwidth available to a flow is then the minimum of its bandwidth shares on the source’s uplink and the sink’s downlink. This allocation algorithm is run on all the flows of a link whenever a flow is added or removed from the link.

While this allocation technique is adequate for simulating many scenarios, it falls short when it comes to simulating nodes with widely varying access link capacities. Let us consider a scenario with three nodes, A, B, and C. Node A has a downlink capacity of 4 Mbps. Node B has an uplink capacity of 1 Mbps, while node C has an uplink capacity of 4 Mbps. We set up two flows for this scenario, one running from B to A, and the other from C to A. Running the minimum-share allocation algorithm, the flow from B to A would receive an allocation of $\min\left(\frac{1\text{Mbps}}{1}, \frac{4\text{Mbps}}{2}\right) = 1\text{Mbps}$, and the flow from C to A would be allocated $\min\left(\frac{4\text{Mbps}}{1}, \frac{4\text{Mbps}}{2}\right) = 2\text{Mbps}$. Note that 1 Mbps worth of A's downlink capacity lays unused, when with real TCP the flow from C to A can comfortably use that up
and run at 3 Mbps. Since we are interested in simulating a heterogeneous mix of access link capacities, such a shortcoming may lead to simulation results that are unnecessarily pessimistic with regards to link utilizations and transfer rates.

```
function otherSide(flow, link)
    if (link == flow.sourceUplink)
        return flow.destDownlink
    else
        return flow.sourceUplink

function allocateBandwidth(link)
    for each flow in link.flows
        otherLink = otherSide(flow, link)
        flowDetails[flow] = record(
            theFlow = flow,
            otherSideFair = otherLink.capacity / otherLink.numFlows,
            otherSideCurrentMax =
                flow.allocatedBandwidth +
                otherLink.unallocatedBandwidth,
            potential = max(otherSideCurrentMax, otherSideFair))
    end

    sort flowDetails by potential in ascending order

    remainingCap = link.capacity

    for (i = 0; i < link.numFlows; i++)
        share = min(remainingCap / (link.numFlows - i),
                    flowDetails[i].potential)
        flowDetails[i].theFlow.allocatedBandwidth =
            min(share, flowDetails[i].otherSideCurrentMax)
        remainingCap = remainingCap - share
    end
```

Figure 4.1: Pseudocode for the maximum-share bandwidth allocation algorithm

To ameliorate this situation, we introduce a more refined allocation algorithm, which we call maximum-share allocation. It picks up where minimum-share leaves off, seeking to put into use any bandwidth capacity left unallocated by
minimum-share. Figure 4.1 lists the pseudocode for this algorithm.

The algorithm starts by calculating the bandwidth potential for each flow, namely the maximum share that the flow can obtain from otherLink, i.e., the link on the other side for the flow. This is the maximum of the equal-split share of bandwidth on otherLink, and the sum of the current allocation plus all of the unallocated slack on otherLink. Then, starting with the flow with the lowest potential, the algorithm iteratively sets aside a share which is the minimum of the flow's potential and the equal split of the remaining capacity. However, rather than assigning this share directly to the flow, we limit it to the maximum bandwidth currently achievable on the link on the other side. In the end, the algorithm may potentially leave some slack on the link. This is intentional, as the slack could then be taken up by the flow later when otherLink runs the algorithm and discovers this usable but unallocated bandwidth on this link.

If we run the maximum-share algorithm on the preceding scenario, we will get the desired result, namely that the flow from C to A will be allocated 3 Mbps of bandwidth on both C's uplink and A's downlink.

4.1.2 Evaluation Metrics

We summarize the metrics we use to evaluate the effectiveness and performance of our helpers mechanism in Table 4.1.

As the main objective of our design is the effective utilization of bandwidth resources, especially those provided by helpers, our primary metric for measuring performance is that of aggregate network throughput. In particular, we are interested in the rate at which blocks are delivered to regular peers, which we call useful throughput. In addition, we also record peers' download times, i.e., how long it takes for a peer to completely download the file. From the perspective of peers in real peer-to-peer networks, this is in fact the most important performance metric.

Moreover, we quantify the helpers' contribution by measuring, over time,
Table 4.1: Evaluation Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Short Name</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) # blocks delivered to regular peers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) aggregate download rate of regular peers</td>
<td>useful throughput</td>
<td>time derivative of (1)</td>
</tr>
<tr>
<td>3) # blocks delivered to helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4) aggregate helper download rate</td>
<td></td>
<td>time derivative of (3)</td>
</tr>
<tr>
<td>5) # blocks uploaded by helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6) aggregate helper upload rate</td>
<td></td>
<td>time derivative of (5)</td>
</tr>
<tr>
<td>7) (# blocks transferred from helpers to regular peers) - (# blocks transferred from regular peers to helpers)</td>
<td>cut flow</td>
<td></td>
</tr>
<tr>
<td>8) download time for regular peers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9) aggregate upload rate aggregate upload capacity</td>
<td>uplink utilization</td>
<td></td>
</tr>
<tr>
<td>10) aggregate download rate aggregate download capacity</td>
<td>downlink utilization</td>
<td></td>
</tr>
</tbody>
</table>

the number of blocks both downloaded and uploaded by helpers, the utilization levels of links, as well as the net flow of blocks through the cut separating helpers from regular peers (which we call the cut flow).

4.1.3 Baseline Test Cases

In order to get a better sense of the performance gains achieved by the addition of helpers, we set up three baseline cases for comparison in each of the scenarios presented. The first baseline (denoted NoHelpers) is the case where there are no helper nodes in the system, leaving only the initial seed and the regular peers. This represents a base performance level, upon which the addition of helpers is expected to make an improvement.
The second case (SeedHelpers) is where all the helper nodes join the system as seeds. This implies that the helpers are maximally equipped to provide blocks wanted by other regular peers. Results obtained for this case will be a theoretical limit to the effectiveness of the helper mechanism, as seeds are in fact perfect helpers.

The third and final baseline (FakeHelpers) has regular nodes masquerading as helpers. In other words, this test case is the same as the primary test case with real helpers (denoted Helpers) in all aspects, except with the rate-limiting mechanism turned off. These fake helpers will therefore try to download blocks as fast as they can. This represents an attempt to improve the performance perceived by regular peers by simply throwing in more regular peers into the system.

### 4.2 Experiments

#### 4.2.1 Setup

Our swarming system and helpers mechanism are configured by a number of parameters, as discussed in Chapter 3 and summarized in the Glossary (Appendix A). For the experiments presented in this section, we have a set of default settings for these parameters, as presented in Table 4.2. These settings will be used for a particular experiment unless explicitly overridden. The regular peers all connect via asymmetric access links. This makes it likely that the demand for bandwidth resources will be greater than the supply, and hence making the deployment of helpers worthwhile. We choose to have all nodes complete their downloads, as premature departures would have different impact on different but related test cases, rendering it more difficult to compare results across these cases. A similar motivation is behind the choice of setting $r_{uninvite}$ to 0.

In all of the experiments, we have configured the setup such that all the available candidate peers end up being invited to join the swarm as helpers. There-
Table 4.2: Default simulation parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>File size:</td>
<td>128MB</td>
</tr>
<tr>
<td>Block size:</td>
<td>256KB</td>
</tr>
<tr>
<td>Initial seed's link</td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td>2Mbps up, 2Mbps down</td>
</tr>
<tr>
<td>Latency:</td>
<td>10ms</td>
</tr>
<tr>
<td>Regular peers' links</td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td>200Kbps up, 2Mbps down</td>
</tr>
<tr>
<td>Latency:</td>
<td>Gaussian distributed, with $\mu = 50ms, \sigma = 20ms$</td>
</tr>
<tr>
<td>$N^d_A$:</td>
<td>$\max\left(\frac{\text{Pcap}}{200\text{Kbps}}, 5\right)$</td>
</tr>
<tr>
<td>$N^s_A$:</td>
<td>$\max\left(\frac{\text{Ucap}}{200\text{Kbps}}, 5\right)$</td>
</tr>
<tr>
<td>$k_{penalty}$:</td>
<td>0.875</td>
</tr>
<tr>
<td>$k_{thres}$:</td>
<td>0.0001</td>
</tr>
<tr>
<td>$k_{upload}$:</td>
<td>0.6</td>
</tr>
<tr>
<td>$\tau_{treesal}$:</td>
<td>30 seconds</td>
</tr>
<tr>
<td>Duration of a peer's stay:</td>
<td>T minutes after it completely downloads the file, T exponentially distributed, with $\lambda = \frac{1}{60 \text{minutes}}$</td>
</tr>
<tr>
<td>$r_{invite}$:</td>
<td>1.25</td>
</tr>
<tr>
<td>$r_{uninvite}$:</td>
<td>0</td>
</tr>
<tr>
<td>$k_{helper}$:</td>
<td>4</td>
</tr>
</tbody>
</table>

4.2.2 A First Scenario: Instantaneous Flash Crowd with Broadband Peers

We begin our presentation of experiments and results with a case study, through which we intend to demonstrate the effectiveness of helpers, and to take a closer
look at the inner workings of our helpers mechanism.

In this first scenario, we investigate the ability of helpers in mitigating the effects of a large spike in the number of requests for a file by having all peers join the system simultaneously. Listed in Table 4.3 are the simulation parameters for this scenario.

Table 4.3: Simulation parameters for the first scenario

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of regular peers</td>
<td>instantaneous</td>
</tr>
<tr>
<td>Number of regular peers</td>
<td>1000</td>
</tr>
<tr>
<td>Number of helpers</td>
<td>2000</td>
</tr>
<tr>
<td>Helpers' links</td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td>200Kbps up, 2Mbps down</td>
</tr>
<tr>
<td>Latency:</td>
<td>Gaussian distributed, with $\mu = 50ms$, $\sigma = 20ms$</td>
</tr>
</tbody>
</table>

For this experiment, we have 1000 peers joining the swarm at $t = 1s$. As well, we have 2000 helpers on hand ready to be added to the swarm. These helpers are connected by asymmetric access links, each with an upload capacity of 200Kbps and download capacity of 2Mbps. In total, they could provide up to an additional 400Mbps of upload capacity to the swarm.

Figures 4.2 and 4.3 respectively show the number of blocks received by regular peers over time and the corresponding useful throughput. As we can see, the line for our primary test case with real helpers, identified as Helpers, falls between the lines for NoHelpers and SeedHelpers in both graphs. This signifies that the helpers indeed made a positive contribution to the swarm by allowing regular peers to download blocks at a faster rate. Examining Figure 4.3 more closely, we find that the useful throughput achieved by Helpers is around 400Mbps, and is 200Mbps more than the useful throughput in NoHelpers. Out of the 400Mbps made available by the helpers, it turns out that half of it is contributed to the benefit of regular peers. The remaining 200Mbps goes to support the helpers' own acquisition of blocks.
We also note that the line for FakeHelpers in both graphs falls below that of NoHelpers, confirming our expectation that the addition of untuned regular peers can potentially drain resources away from serving those peers that actually
want the file. To confirm this, we plot the cut flow for **Helpers** and **FakeHelpers** in Figure 4.4, where we define the cut flow to be the number of blocks transferred from helpers to regular peers, minus the number of blocks transferred from regular peers to helpers. Whereas the cut flow for **Helpers** grows as time progresses, the cut flow for **FakeHelpers** remains in negative territory throughout the entire simulation, and is in fact decreasing for the majority of the time. That means there are more blocks transferred from regular peers to the fake helpers than blocks transferred in the opposite direction, representing a net loss to the regular peers as a whole.

From the end user's point of view, perhaps nothing matters more than how long it takes for the download to complete. In Table 4.4, we summarize the average download times obtained in the primary test case **Helpers** and in the baseline cases **NoHelpers**, **SeedHelpers**, and **FakeHelpers**.

Through their participation, the helpers are able to reduce the average download time of regular peers by 1997 seconds. On the other hand, the fake helpers
Table 4.4: Average download times for the first scenario

<table>
<thead>
<tr>
<th>Test case</th>
<th>SeedHelpers</th>
<th>Helpers</th>
<th>NoHelpers</th>
<th>FakeHelpers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average download time</td>
<td>1772s</td>
<td>3443s</td>
<td>5540s</td>
<td>5817s</td>
</tr>
<tr>
<td>Speed-up relative to NoHelpers</td>
<td>3.126</td>
<td>1.609</td>
<td>1</td>
<td>0.952</td>
</tr>
</tbody>
</table>

Figure 4.5: Total number of blocks transferred to helpers

of FakeHelpers worsened the performance for everyone, increasing the average download time by 277 seconds.

We have seen thus far that the rate limiting mechanism in helpers has differentiated them from the fake helpers of FakeHelpers and has succeeded in making it possible for helpers to truly help out. To better understand what this mechanism does, we plot the number of blocks downloaded by helpers and fake helpers over time and the corresponding throughput in Figures 4.5 and 4.6. In between the ramp-up and ramp-down periods of the system, namely in the time interval \( t = 1500s - 2500s \), the download rate of helpers is a mere 40% of the download rate of the fake helpers.
Figure 4.6: Aggregate helper download rate

Figure 4.7: Aggregate helper upload rate

Figure 4.7 shows the aggregate upload rate of all the helpers in Helpers. Comparing this graph with the download rate shown in Figure 4.6, we observe that the download rate during the stable period of $t = 1500s - 2500s$ is about 40% of
the upload rate. In other words, every block downloaded by a helper is uploaded 2.5 times on average, which is close to our target upload factor of $k_{\text{upload}} \cdot N_A^u = 0.6 \cdot 5 = 3$.

### 4.2.3 Varying Arrival Rates

Our first scenario focused upon an instantaneous flash crowd. In order to more closely model the dynamics of a real flash crowd, we have also conducted tests where peers join the swarm one at a time at particular arrival rates. We summarize the setup for these simulations in Table 4.5.

<table>
<thead>
<tr>
<th>Arrival of regular peers</th>
<th>governed by Poisson process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected arrival rate</td>
<td>10/minute, 20/minute, 40/minute</td>
</tr>
<tr>
<td>Time interval of arrivals</td>
<td>from $t = 0s$ to $t = 10000s$</td>
</tr>
<tr>
<td>Number of regular peers</td>
<td>1675, 3341, 6719</td>
</tr>
<tr>
<td>Number of helpers</td>
<td>2000</td>
</tr>
<tr>
<td>Helpers' links</td>
<td>Capacity: 200Kbps up, 2Mbps down</td>
</tr>
<tr>
<td></td>
<td>Latency: Gaussian distributed, with $\mu = 50ms, \sigma = 20ms$</td>
</tr>
</tbody>
</table>

As in the previous section, we are interested in comparing the performance of our helpers-enabled swarming system with the baseline cases. For this comparison, we focus our attention to the scenario where the arrival rate is 20 regular peers per minute.

In Figure 4.8, we observe that our swarm with 2000 helpers (indicated by Helpers) is able to deliver close to the same number of blocks to regular peers as the swarm with 2001 seeds (indicated by SeedHelpers), especially after a warm-up period in the first 4000s. The useful throughput graph in Figure 4.9 confirms that both swarms achieve virtually identical performance after $t = 4000s$.

In contrast, FakeHelpers's swarm with fake helpers performs poorly in the
first 6000s, lagging behind the **NoHelpers** baseline in terms of number of blocks delivered to regular peers. Past $t = 6000s$, however, it starts to pick up the pace, and ultimately by $t = 8000s$, it is able to catch up to the **SeedHelpers** baseline and
surpasses our helpers-enabled swarm. Looking at it in a different perspective, we observe in Figure 4.9 that \textbf{FakeHelpers} is able to sustain a higher useful throughput than the other three cases after $t = 6000s$. To help see why this is the case, let us further investigate the performance characteristics of the swarm in \textbf{FakeHelpers}.

Figure 4.10 plots the number of blocks delivered to the helpers of \textbf{Helpers} and the fake helpers of \textbf{FakeHelpers}. In it, we can see that by $t = 6000s$, most of the fake helpers of \textbf{FakeHelpers} have downloaded the entire file and have become seeds. The subsequent graph (Figure 4.11) then displays, for \textbf{FakeHelpers}, the respective aggregate download rates for regular peers and fake helpers, as well as the total upload capacity of the swarm. Indeed, past $t = 6000s$, the download rate of the fake helpers falls off rapidly, and the entire swarm’s upload capacity is then devoted to serving regular peers. This accounts for the spike in performance originally observed in Figure 4.8.

Coming to the metric of download times, we see from the graphs in Figures 4.12 and 4.13 that helpers are able to lower the download time for the peers
that join the swarm early. The first completions occur just before $t = 4000s$, and after that peers complete their downloads in about $1500s$. On the other hand, while the fake helpers do contribute to lower download times, the first download com-
pletions occur only after $t = 6000s$, and not until $t = 8000s$ do peers begin to complete in 1500s. Thus, we have shown that our rate-limiting mechanism is successful in enabling helpers to mitigate the effects of flash crowds early on.

Figure 4.13: Download time vs. start time

Figure 4.14: Total number of blocks transferred to regular peers
To study the effects of different arrival rates, we also ran our experiments for the arrival rates of 10 and 40 peers per minute (in addition to the aforementioned case of 20 peers per minute). Our plots of the throughput levels in Figures 4.14 and 4.15 show that our helpers mechanism behaves consistently across different arrival rates, modulo a scaling factor due to the difference in the number of peers in the swarm.

As with the instantaneous arrival scenario of the previous section, we see in Figure 4.16 that our helpers (on asymmetric access links) are able to attain good levels of cut flow, while the fake helpers of FakeHelpers trail behind substantially for all arrival rates.

Figure 4.17 presents the aggregate upload and download rates of the helpers in the three scenarios. As the arrival rate increases, the aggregate upload rate of the 2000 helpers also increases, while the aggregate download rate decreases. To see why this is the case, we note that each helper halts downloading further blocks as long as it has a certain number of unfulfilled blocks. With a higher arrival rate comes a higher connection request rate for each helper. In turn, due to the
connection selection algorithm, each helper is more likely to be connected more often to peers with very few blocks in common with the helper. First of all, this means that the helper would have more to offer to its sinks, reducing connection
idle times. At the same time, the helper would be able to upload more fulfilled blocks to its sinks, slowing down the delisting of unfulfilled blocks, and ultimately reducing the rate at which the helper downloads new blocks. Therefore, a higher arrival rate actually improves the operational efficiency of the helpers mechanism.

4.2.4 Varying Helpers’ Bandwidth Characteristics

Having shown that our helpers mechanism is effective in enabling peers on asymmetric access links to lend help to a swarm, we now explore the implications of having well-endowed helpers connected to the network via fast, symmetric access links. In particular, it remains to be seen whether the rate-limiting nature of our mechanism is suitable for these symmetric links, where there is less of a need to throttle a helper’s download rate to be no higher than its upload rate. The configurations of our experiments are listed in Table 4.6. Here, we use scenarios involving instantaneous as well as phased arrivals. With regards to the helpers, the three cases are: 1) all on asymmetric links, 2) all on symmetric links, and 3) half asymmetric, half symmetric links. In all these scenarios, the number of helpers is chosen such that the helpers’ aggregate upload capacity is fixed at 400Mbps.

As the relative proportion of helpers on symmetric links increases, the throughput levels of the FakeHelpers baseline come closer to those of the helpers-enabled Helpers. This result is shown in Figures 4.18(a) and 4.18(b) for the simultaneous arrival scenarios, and in Figures 4.19(a) and 4.19(b) for the phased arrival scenarios. At the extreme of having all helpers on symmetric links, the fake helpers of FakeHelpers can perform just as well as the helpers, but never significantly exceeding the performance of real helpers. Even though the rate-limiting mechanism may seem restrictive for symmetric links, it nonetheless does not hamper the helpers on such links from effectively utilizing their upload capacities in serving other peers.

Indeed, as Figures 4.20 and 4.21 show, the rate-limiting mechanism continues to function according to its design, slowing down the helpers’ download
Table 4.6: Simulation parameters for varying the mix of helpers

<table>
<thead>
<tr>
<th>Arrival of regular peers</th>
<th>governed by Poisson process</th>
<th>instantaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected arrival rate</td>
<td>20/minute</td>
<td>—</td>
</tr>
<tr>
<td>Time interval of arrivals</td>
<td>from $t = 0s$ to $t = 1000s$</td>
<td>—</td>
</tr>
<tr>
<td>Number of regular peers</td>
<td>3341</td>
<td>1000</td>
</tr>
<tr>
<td>Number of helpers</td>
<td>2000 asy.</td>
<td>2000 asy.</td>
</tr>
<tr>
<td></td>
<td>200 sym.</td>
<td>200 sym.</td>
</tr>
<tr>
<td></td>
<td>1000 asy.</td>
<td>1000 asy.</td>
</tr>
<tr>
<td></td>
<td>100 sym.</td>
<td>100 sym.</td>
</tr>
<tr>
<td>Helpers’ links</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asymmetric</td>
<td>200Kbps up, 2Mbps down</td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency:</td>
<td>Gaussian distributed, with $\mu = 50ms, \sigma = 20ms$</td>
<td></td>
</tr>
<tr>
<td>Symmetric</td>
<td>2Mbps up, 2Mbps down</td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency:</td>
<td>Gaussian distributed, with $\mu = 50ms, \sigma = 20ms$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.18: System throughput results for instantaneous arrival scenarios

(a) # blocks transferred to regular peers

(b) Useful throughput levels
Figure 4.19: System throughput results for phased arrival scenarios

(a) # blocks transferred to regular peers  (b) Useful throughput levels

Figure 4.20: Helper efficiency results for instantaneous arrival scenarios

(a) # blocks transferred to helpers  (b) Aggregate helper download rate
progress appropriately regardless of their link capacities.

4.2.5 *Varying the Number of Helpers*

In the previous sections, we have investigated the performance of our mechanism in settings with various arrival rates and different mixes of helper capabilities. From these scenarios, we pick four representatives, as listed in Table 4.7 to form a test suite. To facilitate communication, these scenarios will be referred to by their names, i.e., $sym_{20}$, $asym_{20}$, $sym_{\infty}$ and $asym_{\infty}$. We will be using this test suite to evaluate the effects of other simulation parameters.

One parameter we are interested in is the number of helpers in the system. As we have seen in Section 4.2.3, an increase in the peer arrival rate makes the helpers function more efficiently by providing more blocks while downloading less. We should note that the higher the arrival rate, the higher the number of active peers in the swarm. Therefore, there is potentially a more fundamental explanation to the increased helper efficiency, namely that there is a link between
Table 4.7: Simulation parameters for the test suite

<table>
<thead>
<tr>
<th>Name</th>
<th>$sym_{20}$</th>
<th>$asym_{20}$</th>
<th>$sym_{\infty}$</th>
<th>$asym_{\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of regular peers</td>
<td>governed by Poisson process</td>
<td>instantaneous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected arrival rate</td>
<td>20/minute</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Time interval of arrivals</td>
<td>from $t = 0s$ to $t = 10000s$</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Number of regular peers</td>
<td>3341</td>
<td>1000</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Helpers' links</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asymmetric</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td>200Kbps up, 2Mbps down</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Latency:</td>
<td>Gaussian distributed, with $\mu = 50ms, \sigma = 20ms$</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capacity:</td>
<td>2Mbps up, 2Mbps down</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Latency:</td>
<td>Gaussian distributed, with $\mu = 50ms, \sigma = 20ms$</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

the relative proportions of helpers and regular peers in a swarm and the efficiency of the helpers.

To validate this hypothesis, we ran the four scenarios in the test suite with fewer helpers, at levels that are $3/4$, $1/2$ and $1/4$ of the original number of helpers (see Table 4.8). In other words, these experiments are run with aggregate helper upload capacities of 300, 200, and 100Mbps.

Table 4.8: Simulation parameters for varying the number of helpers

<table>
<thead>
<tr>
<th>Test suite scenarios</th>
<th>$asym_{20}$, $asym_{\infty}$</th>
<th>$sym_{20}$, $sym_{\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of helpers</td>
<td>500</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1500</td>
<td>150</td>
</tr>
</tbody>
</table>

In order to compare results across different experiments with varying numbers of helpers, we examine the uplink and downlink utilization levels of helpers and regular peers. The uplink utilization of a set of peers is defined to be their aggregate upload throughput divided by their aggregate upload capacity, and the
From Figure 4.22, we see that while the helpers are able to achieve high levels of uplink utilization regardless of the number of helpers in the swarm, there is in fact a small decrease in uplink utilization as the number of helpers increases.
Therefore, a small group of helpers in a big swarm can be just as effectively utilized as a larger group of helpers in a smaller swarm. At the same time, the benefit of having helpers grows with the number of helpers, as demonstrated in Figure 4.23 by the proportionate increase in the downlink utilization levels of regular peers in
all four test-suite scenarios.

4.2.6 Varying $k_{thres}$

As outlined in Section 3.3.2, the helpers mechanism is configured by two main parameters, namely $k_{thres}$ and $k_{upload}$. In order to better understand the effects each of these parameters have on the operation and performance of our mechanism, we simulated the four test-suite scenarios again with varying values for these parameters. In the first set of experiments, we held the $k_{upload}$ value fixed at our default value, and varied the value $k_{thres}$ (see Table 4.9).

![Table 4.9: Simulation parameters for varying $k_{thres}$](image)

Table 4.9: Simulation parameters for varying $k_{thres}$

<table>
<thead>
<tr>
<th>Test suite scenarios</th>
<th>$sym_{20}$, $asym_{20}$, $sym_{\infty}$, $asym_{\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_{thres}$</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

![Figure 4.24: Useful throughput levels for scenarios with helpers on asymmetric links](image)

Figure 4.24: Useful throughput levels for scenarios with helpers on asymmetric links
The useful throughput levels for the scenarios \( \text{asym}_\infty \) and \( \text{asym}_{20} \) are shown in Figure 4.24. They show that, for both scenarios, the useful throughput levels increase as \( k_{\text{thres}} \) decreases. This is in line with the intuition that limiting the download rates of helpers on asymmetric links is crucial to the efficiency of the helpers mechanism. The larger the value of \( k_{\text{thres}} \), the longer a helper stays in its initial ramp-up mode where its download rate is unrestricted, and the more bandwidth resources the helper takes away from the rest of the swarm.

![Figure 4.25](image)

(a) \( \text{asym}_\infty \)  
(b) \( \text{asym}_{20} \)

Figure 4.25: Aggregate helper download rates for scenarios with helpers on asymmetric links

Figure 4.25 plots the aggregate download rate of the helpers. These results confirm that a lower \( k_{\text{thres}} \) value translates to lower helper download rates.

In the other two scenarios \( \text{sym}_\infty \) and \( \text{sym}_{20} \), the situation is reversed. From Figures 4.26 and 4.27, we learn that both the useful throughput levels and the cut flow levels decrease as \( k_{\text{thres}} \) decreases. This is due to the fact that a lower \( k_{\text{thres}} \) means that the helpers go into the rate-limiting mode sooner. For helpers on symmetric links, overly restricting their download rates has the consequence of not allowing them to download a wide enough selection of blocks early on for their
many sinks to request, causing many of the helpers' upload connections to unnecessarily go idle.
From these results, we conclude that there is room for improvement in our setting for the threshold value $T_A$, currently set to $k_{thres} \cdot UC_A$. In particular, the efficiency of helpers could be improved by further lowering $T_A$ when the upload capacity $UC_A$ is low, and further increasing $T_A$ when $UC_A$ is high. Potential solutions for achieving this include using a linear function in $UC_A$ with a negative $y$-intercept, and using a higher-order function.

4.2.7 Varying $k_{upload}$

Having seen and analyzed the effects of changing the value of $k_{thres}$, we complete our study on the rate-limiting equation by focusing on the effects of varying $k_{upload}$. We conducted a set of experiments where we hold $k_{thres}$ constant at its default value, and let $k_{upload}$ range from 0.2 to 0.8 (see Table 4.10). In other words, each block downloaded by a helper must be uploaded on 20% to 80% of its outgoing connections before being taken off the list of unfulfilled blocks.

Table 4.10: Simulation parameters for varying $k_{upload}$

<table>
<thead>
<tr>
<th>Test suite scenarios</th>
<th>$sym_{20}$, $asym_{20}$, $sym_{\infty}$, $asym_{\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_{upload}$</td>
<td>0.2 0.4 0.6 0.8</td>
</tr>
</tbody>
</table>

As the results in Figures 4.28 and 4.29 show, a higher value of $k_{upload}$ has the effect of lowering the download rates on helpers. Therefore, for the scenarios where having lower helper download rates is important, namely those involving helpers on asymmetric links, the useful throughput levels do indeed increase with $k_{upload}$. On the other hand, varying the value of $k_{upload}$ has a less pronounced effect on the useful throughput levels for the scenarios $sym_{\infty}$ and $sym_{20}$ involving helpers on symmetric links.
Figure 4.28: Useful throughput levels and aggregate helper download rates for scenarios with helpers on asymmetric links.
Figure 4.29: Useful throughput levels and aggregate helper download rates for scenarios with helpers on symmetric links
4.2.8 Summary

We have now shown that our helpers mechanism succeeds in effectively utilizing the bandwidth resources of helpers in a wide variety of scenarios. In Section 4.2.2, we saw that helpers on asymmetric links are able to provide significant benefits in the case where a flash crowd arrives at an instant, by increasing useful throughput levels and lowering download times. On the other hand, the unregulated fake helpers of the corresponding FakeHelpers setup actually drain bandwidth resources away from regular peers, lowering the useful throughput levels and slowing down all downloads. As the only difference between the Helpers and FakeHelpers setups is our adaptive rate-limiting mechanism, these contrasting results demonstrated the importance of this mechanism in making helpers work.

We explored the effects of having peers arrive one at a time, with various arrival rates, in Section 4.2.3. In those scenarios, our helpers are able to increase the useful throughput levels early on, and very quickly get to a point where they are operating almost as efficiently as the fully-seeded helpers of SeedHelpers. Once again, the fake helpers take much longer to arrive at a comparable level of efficiency.

We varied the bandwidth characteristics of helpers in Section 4.2.4. The scenarios involved different mixes of helpers on symmetric and asymmetric access links. The results showed that our adaptive rate-limiting mechanism does not hinder those helpers that are able to upload as fast as they can download.

In Section 4.2.5, we varied the number of helpers in the swarms, and found that helpers are able to put their uplink capacities into good use regardless of the number of helpers. In fact, the helpers' efficiency increases slightly as the number of helpers decreases.

For our final two sets of experiments, we configured our helpers mechanism with different values of the parameters $k_{\text{thres}}$ and $k_{\text{upload}}$. The results in Section 4.2.6 showed that a higher value of $k_{\text{thres}}$ is beneficial for helpers with well-
endowed uplinks, while the reverse is true for helpers with slow uplinks. Therefore, there is room for improvement in our current setting of the threshold value $T_A$ by the simple equation of $T_A = k_{thres} \cdot UC_A$. In Section 4.2.7, we found that increasing the value of $k_{upload}$ lowers the download rates of helpers while improving the useful throughput levels. This measure is especially effective for helpers on asymmetric access links.
CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

In the client-server model predominant in today's Internet, the load on a server has a significant impact on the level of service perceived by its clients. In particular, given that a server is connected to its clients via an access link of finite capacity, the data transfer rate to each of its clients drops inversely as the number of clients served by the server. This limitation significantly hampers the scalability of such a setup, especially when long-lived connections are involved in the transfer of large files. If such a high load can be anticipated in advance, it is possible to alleviate this problem by provisioning additional upload capacity. However, it is now commonplace for a particular piece of content on the Internet to experience a sudden spike in popularity and hence also in request rate, only to have the number of requests returning to normal levels in a few hours or a few days' time. Such an occurrence, called a "flash crowd" in Internet parlance, is difficult to address via adjusting the provisioning level alone, as it may come and go so quickly for administrative actions to be taken. Moreover, the operators of the server may not have the financial means to scale up the bandwidth of their access link.

To mitigate the problem of the flash crowd, many solutions have been de-
developed, some involving the service provided by additional servers from around the world (mirrors and CDNs), while others involving the cooperation and collaboration of the clients themselves. In this thesis, we explored the space of collaborative solutions, and found that existing systems, such as multicast streaming and swarming, are often able to adequately deliver large amounts of content to participating peers in a scalable way. However, these systems use a tit-for-tat notion of fairness, and thus are not well-suited for harnessing the bandwidth resources of idle peers that are willing to contribute.

Leveraging some insights in the helpful nature of interior nodes in multicast trees, we have developed a novel augmented swarming system that allows peers to contribute their bandwidth resources even when they are not interested in the content being disseminated in the swarm. Our design consists of a helpers mechanism which ensures that a helper’s participation in a swarm is always to the benefit of others. It achieves this objective by employing an adaptive rate-limiting mechanism, so that the helper is able to maximize its uplink utilization while downloading as little and as slowly as possible.

For our evaluation, we have run a diverse set of experiments, testing the helpers mechanism under a variety of conditions, including varying arrival rates, varying numbers of and provisioning levels of helpers, and different parameter settings of the mechanism itself. The experiments were run on a flow-level simulator of our own design, whose strength lies in its ability to deal with heterogeneous mixes of link capacities. Our results show that helpers are effective in contributing their bandwidth resources under all circumstances, and always perform at a level comparable to, and often much better than, the naïve addition of untuned peers to the swarm.

Using our framework, a content provider would be able to invite additional helpers into its swarm to aid in the handling the increased load from a flash crowd. At the same time, the flexibility of the helpers mechanism enables the development
of a peer-to-peer content delivery infrastructure where the peers always have the ability and opportunity to contribute, regardless of whether or not these peers are actively downloading content of their own desires.

5.2 Future Work

With regards to our helpers mechanism, we have identified four directions for future work. First, there is room to further improve the efficiency of helpers by fine-tuning the relevant parameters such as $k_{\text{thres}}$ and $k_{\text{upload}}$. In addition, it would be interesting to study how well the idea of rate-limited helpers would work in an existing system such as BitTorrent, where connection evaluations are made primarily based on the attained transfer rate on each connection. Thirdly, we would like to broaden our scope and investigate how our mechanism performs within other possible swarming systems as outlined in our taxonomy, such as ones where peers could be located based on which portions of the content they contain. For example, a group of helpers may each contain a different portion of a file, and the group management mechanism (e.g., a tracker or a DHT substrate) would then be able to direct a peer to the appropriate helpers depending on which blocks it is missing. Finally, we would like to perform our evaluations again in an emulation framework to obtain results which would better represent real-world performance.

Our helpers, by definition, find no value in the content that they help to transfer. An interesting open issue related to this is the design of a suitable incentive mechanism or peer-to-peer economy for rewarding helpers for their contributions. Such a mechanism must be able to resist attacks and cheating attempts from individual malicious nodes as well as nodes in collusion.

Since a helper is often able to effectively utilize its upload capacity while having only slowly downloaded a few blocks, one may consider fixing a hard limit on the number of blocks a helper would cache at any point in time. Interesting issues arising from such a restriction includes the decision of which blocks to evict
when the cache is full, and the corresponding effect on the helper's uplink utilization. Going further down this track, a helper handling multiple files simultaneously could dynamically change the limits on the per-file caches to adapt to changes in the relative popularity of these files. Strategies for doing so must take into account the block-eviction policy adopted.
Bibliography


A.1 Terms

Candidate: a peer that is willing to join a swarm to contribute its bandwidth resources to the aid of others.

Helper: a peer that has joined a swarm only to contribute its bandwidth resources. A helper is not interested in the content that it helps to distribute.

Leecher: a peer with only a partial copy of the file.

Regular peer: a non-helper peer in the swarm (see Helper).

Seed: a peer possessing all the blocks of the file.

Swarm: a set of peers which are uploading and downloading the same file via swarming.

Swarming: a content delivery technique where peers upload and download parts of the content to each other in parallel through multiple connections.

Reevaluation (of unfulfilled blocks): a maintenance routine run periodically by our mechanism to make sure that unfulfilled blocks remain fulfillable, so that no blocks are stuck in the unfulfilled state indefinitely.
Tracker: a server process responsible for coordinating the membership of the swarm.

Unfulfilled block (of a helper peer A): a block possessed by A which has been uploaded less than \( u_{fA} \) times (see \( u_{fA} \)).

Upload factor: the number of times a helper must upload each block it downloads.

### A.2 Notations

\( \Phi_A \): the set of unfulfilled blocks possessed by peer A, defined as \( \Phi_A = \{i | u_A(i) < u_{fA} \} \).

\( \phi_A \): the number of unfulfilled blocks possessed by peer A, defined as \( \phi_A = |\Phi_A| \).

\( \tau_{reeval} \): the time period between successive reevaluations (see Reevaluation).

\( B_A \): the set of blocks possessed by peer A.

\( DC_A \): the download capacity of the peer A.

\( D/S \): the demand/supply ratio of a swarm, calculated as \( D/S = \frac{\sum_{i \in \text{regular peer}} DC_i}{\sum_{i \in \text{all peers}} UC_i} \). It is used by our mechanism to determine whether additional helpers should be invited into the swarm, and whether existing helpers could be asked to leave.

\( desire_A(i) \): the number of A’s sinks which do not have the block \( i \).

\( k_{helper} \): a system-wide parameter dictating the maximum number of helpers allowable in a swarm. In our mechanism, we set the limit to be \( k_{helper} \) times the number of regular peers in the swarm.

\( k_{penalty} \): a system-wide parameter between 0 and 1 used to bias our swarming system towards preferring existing connections over new ones. The bias increases as the value of \( k_{penalty} \) decreases towards 0.
\(k\text{thres}\) : a system-wide parameter dictating the value of \(T_A\) for each helper peer \(A\).
In our mechanism, we set \(T_A = k_{\text{thres}} \cdot UC_A\).

\(k_{\text{upload}}\) : a system-wide parameter dictating the value of \(u_A\) for each helper peer \(A\).
In our mechanism, we set \(u_A = k_{\text{upload}} \cdot N_A^u\).

\(N_A^d\) : the upper limit for the number of incoming (download) connections for peer \(A\).

\(N_A^u\) : the upper limit for the number of outgoing (upload) connections for peer \(A\).

\(r_{\text{invite}}\) : an upper threshold for the value of \(D/S\). When \(D/S > r_{\text{invite}}\), the tracker attempts to invite additional helpers to join the swarm until either \(D/S\) is restored to be below \(r_{\text{invite}}\), or when the number of helpers in the system hits an upper limit.

\(r_{\text{uninvite}}\) : a lower threshold for the value of \(D/S\). When \(D/S < r_{\text{uninvite}}\), the tracker can ask existing helpers in the swarm to leave. It may do so until \(D/S\) is restored to be above \(r_{\text{uninvite}}\).

\(T_A\) : an upper threshold for the value of \(\phi_A\). When \(\phi_A > T_A\), our mechanism halts the helper peer \(A\) from downloading further blocks.

\(u_A(i)\) : the number of times peer \(A\) has uploaded block \(i\). This upload count can be modified artificially by the reevaluation routine (see \textit{Reevaluation}).

\(u_f_A\) : the \textit{upload factor} for a helper peer \(A\), i.e., the number of times \(A\) must upload each block it downloads.

\(UC_A\) : the upload capacity of the peer \(A\).