Evaluating Performance Tradeoffs in a One-to-Many Peer Content Distribution Architecture

SAMI ROLLINS
Department of Computer Science
Mount Holyoke College
USA
srollins@cs.ucsb.edu

KEVIN C. ALMEROTH
Department of Computer Science
University of California at Santa Barbara
USA
almeroth@cs.ucsb.edu

Abstract

Peer-to-peer (P2P) content exchange has recently gained attention from both the research and industrial communities. The dynamic nature of peer networks and the resource constraints of peer hosts have introduced a number of unique technical challenges that must be addressed to make large-scale P2P content exchange applications more efficient. In this work, we expand our previous work on *Pixie*, an architecture that integrates one-to-many distribution of content and peer networks. *Pixie* uses a jukebox-style scheduling mechanism to provide a valuable data location service. Users can browse a listing of all content scheduled to be distributed across the network thus reducing search overhead. Moreover, *Pixie*'s use of one-to-many content distribution provides additional scalability. Our results indicate that, using *Pixie*, we can significantly reduce the resources required to locate and distribute content in peer networks. The properties *Pixie* embodies will become increasingly important as peer content exchange is extended to support more advanced, and possibly commercial applications.

Keywords: peer-to-peer, multicast, data location

1 Introduction

Peer-to-peer (P2P) content exchange has recently gained attention from both the research and industrial communities. Systems like Napster and Gnutella launched P2P into the spot light while systems like Chord [1] and CAN [2] have gone a step further in terms of supporting reliable and efficient content exchange. The range of applications that fall into the P2P space has exploded. From distributed computation to distributed file storage, any application that supports cooperation between end hosts is often considered P2P. However, as P2P becomes recognized as more than just the latest buzzword, there is a call to identify and solve the technical challenges that are faced in P2P environments. The dynamic nature of peer networks and the resource constraints of peer hosts have introduced a number of unique technical problems. Also, while most early deployments work, they are really only simple prototypes and leave many important problems unsolved.

In this work, we extend our previous work on *Pixie* [3] an architecture that integrates one-to-many distribution of content and peer networks. In *Pixie*, peers join the network and retrieve a *schedule* of content to be distributed. Peers can browse the schedule and choose to take advantage of an already-scheduled distribution. Alternatively, a peer can choose to request that a new distribution be scheduled. At the time a distribution is scheduled to begin, the serving peer distributes content to *all* interested peers using any of the available one-to-many communication technologies. While native multicast is the most efficient strategy, the techniques we describe in this work are independent of the chosen delivery mechanism.

Pixie differs from traditional P2P systems in that it provides a new data location service. Its schedule of content currently being delivered, or about to be delivered, serves as a browsable index. In addition, this schedule can reduce searching overhead, in many cases by more than half. Also, by aggregating client requests and using one-to-many batched content distribution, we can actually reduce client wait time by reducing queuing delays as well as reduce the use of resources such as disk space, distribution time, and bandwidth on the serving peer. In the future, we plan to investigate additional savings which may be gained by using a many-to-many distribution scheme in an unreliable network. However, the focus of this work is to validate the service model *Pixie* supports by investigating the performance tradeoffs involved with using *Pixie* as a data location service.

This paper expands our previous work in two ways. First, we further evaluate the tradeoff between *Pixie* search savings and the overhead incurred by the *Pixie* scheme. Our results indicate that *Pixie* reduces search overhead by over half while incurring a manageable overhead with respect to schedule maintenance. In fact, even with the schedule maintenance overhead, *Pixie* outperforms a Gnutella-style flooding search scheme. Additionally, we address reliability and fault handling in *Pixie*. We discuss *Pixie*'s use of existing reliable, one-to-many data distribution protocols as well as the *Pixie* protocol for handling serving peer faults.

This paper is organized as follows. In Section 2 we define peer-to-peer and explore current solutions to the challenges facing P2P applications. Section 3 presents our architecture. In Section 4, we quantify the benefit of using *Pixie* as a data location service and evaluate the tradeoff between the search savings and schedule maintenance overhead of *Pixie*. In Section 5, we evaluate the benefits of using one-to-many content distribution in *Pixie*. Section 6 expands on our fault handling scheme. We conclude in Section 7.

2 Related Work

In this section, we first look at the current impact of P2P content exchange and then address some of the limitations facing P2P networks.

2.1 P2P Content Exchange

P2P encompasses a huge area, from distributed computing [4] to collaborative applications [5]. Applications such as classroom educational tools that enable users to communicate are often considered P2P regardless of their implementation. Alternatively, tools such as application-layer multicast [6] that are implemented using a P2P model, yet support a variety of applications, also fall within the P2P space. In this work, we focus on P2P content exchange applications. This includes the tools, protocols, and applications that support exchange of content between end users.

Napster's pioneering efforts spawned a number of academic and industrial projects aimed at developing efficient, P2P content exchange applications. The primary use of these applications has been the exchange of MP3 music files. But, factors such as increased disk space and higher bandwidth are enabling exchange of other forms of media such as digital video. As more peers increasingly send more and larger files, a number of challenges become apparent.

2.2 Challenges of P2P Content Exchange

As the range of P2P applications increases, P2P content exchange faces a number of challenges. We classify the set of challenges into three areas: peer discovery and group management, data location, and reliable and efficient content exchange.

2.2.1 Peer Discovery and Group Management

The dynamic, ad hoc nature of peer groups makes it difficult to implement peer discovery and group management algorithms. Centralized solutions largely defeat the purpose of a peer network and can be too restrictive if a centralized infrastructure is not available. On the other hand, distributed solutions generally require a great deal of overhead in terms of state kept about other peers and messaging required to maintain that state.

Discovery and management of peer groups can be implemented using a centralized solution, a distributed solution, or a hybrid solution. Centralized solutions such as those used in Napster, Magi, and Groove are most efficient because peers need not keep state about other peers. Moreover, peers can locate each other with a single request to the centralized directory. The problem with this approach is that it requires a centralized infrastructure. Such an infrastructure may not always be available or may introduce a central point of failure which minimizes the benefit of a P2P system.

Distributed solutions such as Gnutella, FreeNet [7], Chord [1], CAN [2], Tapestry [8], and Pastry [9] generally rely on using a well-known peer to discover the rest of the peer group. However, the group management protocols employed by these solutions are distributed. In Gnutella and FreeNet, a peer keeps track of a constant number of other peers. This is efficient in terms of the state kept at each peer. The problem with the approach is that searching the peer network may be slow.

Chord, CAN, Tapestry, and Pastry represent the second-generation of P2P protocols also known as Distributed Hash Tables (DHTs). In each of these protocols, the network is organized such that peers keep track of a logarithmic number of other peers (with respect to the number of peers in the network). When searching, the protocols can guarantee, or guarantee with high probability, that the desired item can be located in a logarithmic number of peer hops.

Peer discovery and group management is the primary focus of P2P content exchange research. We believe that the research in this area is promising. Therefore, in this work, we focus our attention on the following two aspects of content exchange.

2.2.2 Data Location

The distributed nature of peer networks makes data location a difficult problem. Having a centralized index or catalogue of available content again defeats the purpose of a P2P solution and may not be possible if no centralized infrastructure exists. At the other extreme, a fully replicated index wastes resources at each peer and would be difficult, if not impossible, to maintain in a dynamic environment.

Most of the work on supporting data location in peer networks has focused on on-demand searches for information. Systems like Gnutella and Napster, as well as CFS [10], OceanStore [11], and PAST [12], systems built on top of Chord, Tapestry, and Pastry respectively, allow the user to search for a particular document. The user must know the name of the document prior to requesting it and searching is then performed on-demand. While many of these systems claim to support file system-like functionality, the infrastructures do not support file system-like content location. Providing that kind of support would require the application to keep track of metadata about each user's files. Even so, this facility would not support exchange of content between users.

Users may not always have a target item they wish to download. Our solution provides users with catalogue of content that they can *browse*. The benefits of organizing content into a browsable catalogue include both a more pleasurable user experience as well as a reduction of bandwidth and processing power required when searching for information.

2.2.3 Reliable and Efficient Exchange

End-user peers are inherently resource constrained. Especially when compared to centrally administered servers, end-user devices (e.g., desktops, laptops, or PDAs) are restricted with respect to bandwidth, disk space, processing power, as well as up-time since peers cannot be relied upon to remain connected for any specific length of time. This limitation makes reliable content exchange more challenging in the P2P environment. New and innovative schemes must be employed to provide fast downloads and avoid overloading the resources of peers that store *hot* items.

In the P2P space, techniques for making content exchange more reliable and efficient have relied on replicating data within the network. Most deployed systems such as Napster and Gnutella rely on the assumption that data are inherently replicated throughout the network. First, the user selects the best peer from which to download content. If the download request fails, generally because the other peer is not reachable, the user must try a different peer.

This model begins to break down when *hot* data is stored on only a small number of peers. Especially if the peers are resource-constrained, they may not be able to support multiple simultaneous requests from the remainder of the network. This problem is further exaggerated by the fact that peer networks are often composed primarily of *freeriders* [13,14], peers that are only part of the network long enough to retrieve content from other peers.

As peer networks grow, and as multimedia content becomes larger and consumes more resources such as disk space and bandwidth, a more efficient scheme for exchanging content is required. In this work, we focus on the latter two challenges. *Pixie* addresses the problem of data location by providing a browsable catalogue of popular content available across the network. Moreover, the catalogue caches the location of content making data location more resource-efficient. Additionally, we address the challenge of efficient exchange of content by batching requests for content and servicing hundreds or thousands of requests simultaneously. From the client perspective, this greatly reduces the wait time experienced after issuing a request. From the server perspective, we greatly reduce the resources required at the serving peer including disk space, distribution time, and bandwidth.

3 *Pixie* System Design

To overcome many of the challenges of traditional P2P content exchange systems, we explore using one-to-many content distribution in peer networks. In this section, we discuss the motivation of our work, provide an overview of *Pixie*, and discuss the architecture in more detail.

3.1 The AIS

Our architecture is inspired by the Active Information System (AIS) [15], a near-on-demand architecture to support scalable content delivery. The AIS batches client requests for content and produces a schedule of the content to be disseminated. When a client *tunes in* to the system, the client may choose to receive content already scheduled, or may choose to schedule a new distribution. The tradeoff in this case is the time the user must wait to receive content. Dissemination is done using multicast thus relieving much of the burden on the network.

The AIS batching paradigm is well-suited for P2P content exchange. By batching download requests and distributing content to multiple peers in parallel, we can ease much of the burden placed on the serving peer as well as the network. Additionally, the schedule of content to be distributed acts as a *hot list* catalogue. Users can consult the schedule to browse content available in the network.

Unfortunately, the current design of the AIS is targeted toward centralized, video-on-demand (VoD) style applications [16] and is not well-suited to deployment in a P2P network. We have borrowed the AIS paradigm to create an extended architecture to support efficient, scalable, P2P content exchange. Other solutions such as Virtual Batching [17] propose distributed distribution schemes to support VoD applications. However, *Pixie* differs from the Virtual Batching approach in two main ways. First, we address the challenge of data location by providing a schedule of content to the user. Additionally, our goal is to reduce resource usage across the entire network, not just at the server. Another similar approach is Jungle Monkey [18]. The main contribution of Jungle Monkey is an underlying end-host multicast protocol. *Pixie* could be built on top of such a protocol to provide reliable distribution of content and control messages.

3.2 *Pixie* Overview

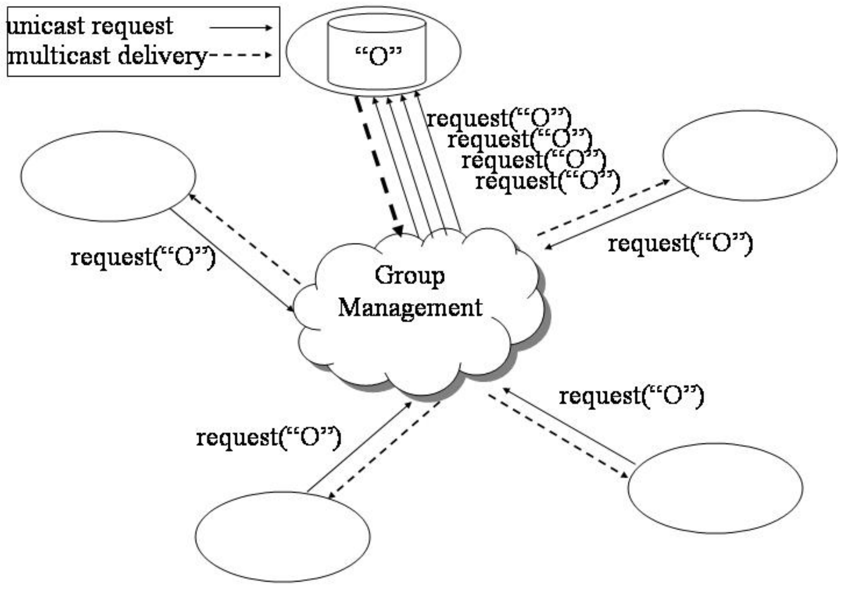

Figure 1: Overlapping requests are aggregated at the serving peer.

Pixie is an architecture which supports one-to-many distribution of content in peer networks. The first goal of *Pixie* is to aggregate peer requests to download content and use intelligent, one-to-many content delivery (e.g., multicast) to enable a large number of peers to take advantage of the same distribution (see Figure 1). The second goal is to publish a schedule of content to be distributed to allow users to *browse* through the most popular subset of content available across the network. *Pixie* can be implemented on top of virtually any peer group management protocol. When a peer joins the network, it requests the *schedule*. The schedule contains information about content that will be distributed (e.g., Gone with the Wind), how the peer is to receive the content (i.e., the IP address of the multicast group), and when the distribution is scheduled to begin (e.g., 8pm GST). If a user is not interested in content already scheduled for distribution, the user may choose to search for and schedule new content. When a new distribution is scheduled, an *updateSchedule* message is propagated to all peers indicating the name of the content that will be distributed, how an interested peer can receive the content, and the scheduled distribution time. At distribution time, interested peers *tune in* to the distribution.

Using this model, peers are able to more rapidly and efficiently locate data of interest. The schedule provides a new service, acting as a browsable hot list of available content within the network. Assuming that many users are interested in the same content, it is likely that a user will find the content he or she is interested in by looking at the schedule, thus easing the burden on the network.

By distributing content using one-to-many distribution, we provide additional scalability properties as well. Efficiency gains come from reducing the load on peers by aggregating requests and servicing multiple peers simultaneously. At the scheduled time, the sending peer distributes the information using one-to-many distribution. All interested peers simply *tune in* and receive the content.

3.3 *Pixie* Architecture

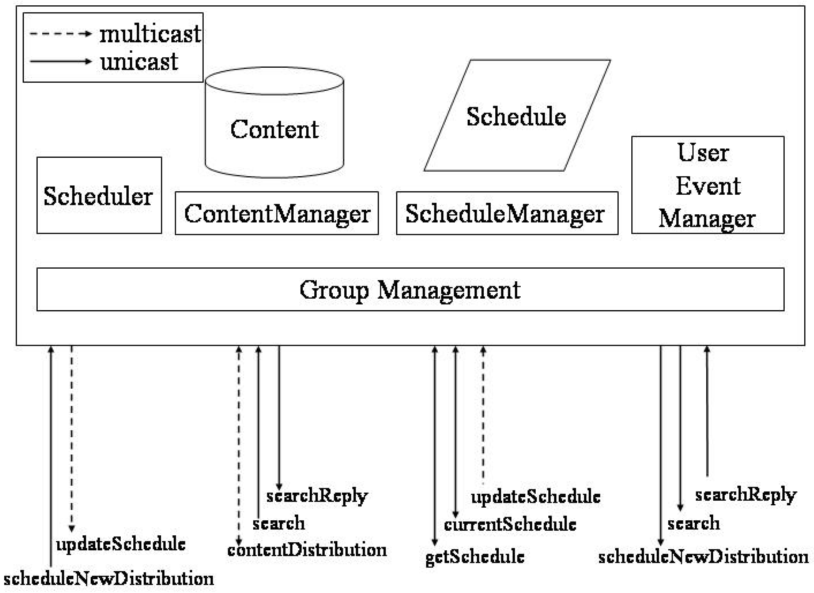

Figure 2: Architecture of a *Pixie* peer.

Figure 2 shows the general architecture of a *Pixie* peer. The *Pixie* components are implemented on top of a group management layer. We place no restrictions on the group management protocol. We envision anything from Napster-style centralized management to Gnutella-style distributed management to Chord-style distributed management. We discuss each component in more detail:

ScheduleManager. The ScheduleManager controls access to the schedule. The schedule contains information about which data are scheduled to be distributed, when distribution will begin, and where the data will be distributed. It is the equivalent of a TV guide that indicates which programs will be showing, at what time, and on which channel. Each peer can retrieve a copy of the schedule when joining the network or can simply join the network and receive any future updates. Where the original copy is found depends on the group management algorithm employed. In a Napster-style network, a getSchedule request will be routed to the centralized server. In a Gnutella-style network, a getSchedule request will be routed to a neighboring peer. We consider the schedule to be *best effort* in that we do not guarantee the peer will receive the latest version. However, if a peer receives a stale version and attempts to search for or schedule an already-scheduled piece of content, the peer serving the content will simply respond with an update indicating where and when the content is already scheduled. The ScheduleManager also receives and applies any updates to the schedule. Schedule updates contain relevant information about newly scheduled distributions (i.e., the content to be distributed, when the distribution will begin, and where the data will be distributed).

Scheduler. The Scheduler handles the scheduling of content distribution for a given peer. When the Scheduler receives a request for new content, it determines when the peer will have the resources available to fulfill the request. For example, if a peer can only support two simultaneous distributions and it is already distributing two streams, the new distribution must wait at least until one of the distributions has finished. The Scheduler may also apply more advanced scheduling algorithms such as delaying distribution in anticipation that more peers will be interested in the same content in the near future. Once the distribution has been scheduled, an updateSchedule message is generated and sent to all peers in the network. The most straightforward method of distributing the updateSchedule message is via multicast (native or application-layer), though a broadcast or gossiping scheme could also be used. In the multicast case, an additional optimization is to use the group abstraction to enable a user to filter out updates for uninteresting content. For example, if a user is not interested in action movies, that user may not subscribe to the multicast group which distributes updates for scheduled deliveries of action movies. An analysis of this optimization is the subject of future work.

In a decentralized system, the Scheduler will exist on each peer and each peer will be responsible for scheduling distribution of its own content. However, a centralized implementation could also be employed. In a Napster-style system, a centralized authority would have information about each peer and could make scheduling decisions based upon that global information. This may be more efficient in terms of resource usage, however would require the presence of a centralized infrastructure.

ContentManager. The ContentManager controls access to the data stored on each peer. If a peer is not interested in scheduled content, it can search the network for other content. Search requests are routed through the network in a manner consistent with the underlying group management protocol. For example, using a Napster protocol, search requests would be routed to a centralized server while if using a Gnutella protocol, requests would be routed to neighboring peers. When a peer receives a search request, the ContentManager consults the content base and responds with information about content matching the search query. In a Gnutella-style network, the search request would then be forwarded to neighboring peers.

The ContentManager is also responsible for distributing content and receiving and storing content distributed by other peers. At the scheduled time, the ContentManager distributes the content, preferably using multicast. While network-layer multicast is the most efficient distribution mechanism, application-layer multicast distribution can be employed for peers without multicast connectivity. A number of appropriate application-layer multicast schemes have recently been developed for the peer-to-peer environment [6,18]. These schemes distribute the burden of content delivery among the participating peers. They offer low overhead for the participating peers and minimal delay with respect to the time to propagate a message from the root to the leaves of the multicast tree.

To ensure reliability, *Pixie* can use a basic reliable multicast distribution scheme for delivery of both content and control messages. However, any straightforward reliable protocol run over either native or application-layer multicast requires receivers to join the distribution from the beginning. Moreover, if a serving peer fails, a new serving peer must start the distribution again from the beginning. To overcome this limitation, we propose and evaluate the use of a digital fountain-style scheme [20]. Using a digital fountain scheme, the ContentManager distributes files that have been encoded using Tornado codes. The serving peer continuously distributes blocks of the encoded file until the client peer has received a sufficient number of blocks to reconstruct the file. Since blocks may be received in any order, a client can join the distribution at any time and take advantage of the distribution in progress. Similarly, if N blocks are needed to reconstruct a file and a serving peer fails after a corresponding receiving peer has received N-X blocks, the receiving peer can join a new distribution and will only need to receive the remaining X blocks.

Using this scheme, a peer can potentially remain continuously occupied, distributing the same file. In the most extreme case, a serving peer distributing a file that requires N blocks to decode will receive a new request for the file after almost all blocks have been sent. In some cases, this behavior may be desirable. However, a peer that stores multiple pieces of popular content may need to perform some form of internal load balancing to ensure that it can service requests for multiple pieces of content. We leave the details of this scheme as future work.

UserEventManager. The UserEventManager processes events from the user and interacts with the user interface. It initiates searches for content specified by the user, requests new content be scheduled, and receives and displays search responses. This component is quite flexible and can be implemented to suit the preferred user interface.

4 Evaluation of *Pixie* Data Location

In this section, we evaluate the benefit of using *Pixie* as a data location service in a peer network. First, we look at the metrics we evaluate and the setup of our experiments. Then, we look at the results of our experiments that evaluate the tradeoff between the search savings gained by using *Pixie*, and the additional overhead of *Pixie* updateSchedule messages. Our conclusion is that, especially in flooding networks like Gnutella, using *Pixie* requires fewer search messages while incurring minimal overhead.

4.1 Metrics

To evaluate the benefit of *Pixie*, we are interested in three primary metrics:

1. **Found** – Found describes how often the user is interested in a particular item and is able to find that item in the schedule. This metric provides us with an idea of how useful the schedule abstraction is from the user perspective.
2. **Number of Search Messages Processed –** Number of Search Messages Processed provides a quantification of the search overhead incurred by *Pixie* versus the overhead incurred using a straightforward search scheme.
3. **updateSchedule Message Overhead** – updateSchedule Message Overhead shows the overhead of distributing updateSchedule messages throughout the network. This metric provides an idea of the cost associated with using the *Pixie* scheme.

4.2 Setup

To evaluate these metrics, we have simulated the *schedule* portion of our architecture. When a request is made, it is processed according to the following algorithm:

```
if the requested item is scheduled
  record as found
  if the distribution has started
    schedule at end time of current distribution
    send updateSchedule message to entire network
else
  send search message
  schedule at current time + 1 minute delay
```

In this experiment, we assume distribution is done through a basic, reliable one-to-many distribution service such that users can only join the distribution from the beginning. An item remains in the schedule from the time it is scheduled until it has been distributed.

This model does not entirely capture two cases. First, we do not capture the case when scheduling incurs an additional delay because a peer's resources are otherwise occupied. However, we claim that the model we use is, in fact, the most restrictive for the metrics we consider. Lower delay means that items remain in the schedule for a shorter period of time and are less likely to be *found*. Incurring an additional delay because a peer is distributing other content or is otherwise busy would only improve our results. Additionally, we do not consider a many-to-many scheduling scheme. However, if such a scheme were to use intelligent updateSchedule propagation techniques, it would not affect any of the metrics we consider here.

Min Time (sec)	Max Time (sec)	Description
1	500	Fast Connection/ High Variance
10	50	Fast Connection/ Low Variance
3800	4300	Mid Connection
10800	21600	Slow Connection/ High Variance
15120	16920	Slow Connection/ Low Variance
120	180	Typical of Current Usage

Table 1: Object distribution times.

To model user behavior, we generate a trace of requests using a Zipf distribution [21]. Recent studies have shown this to be typical for current P2P systems[1]. Unless otherwise noted, we assume a network size of 15,000 peers, use a catalogue of 400,000 items, and run the experiment for a simulated period of 8 hours. We have also run simulations over a simulated period of 24 hours and observed similar results. To analyze the behavior of the system, we vary three main parameters:

1. **Load** – We look at the system behavior under different load conditions by varying the number of requests per second made across the network from 20-90. Values are taken from recent studies of the Gnutella network [14, 22] which indicate that a single peer services or routes roughly 20 requests per second.
2. **Peer Characteristics** – We look at the behavior of the system based on different peer characteristics by varying the time it takes to

[1] http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html

distribute a single object (see Table 1). Small values for the distribution time can be the result of a fast connection or a small object. A large disparity between the min and max times is the result of highly varying peer characteristics. Each distribution time is chosen randomly between the minimum and maximum times.

3. **Network Size** – We look at the system behavior as the network size (e.g., number of participating peers) varies. We look at a small network of 500 nodes, a moderately sized network of 15,000 nodes, and a large network of 50,000 nodes.

4.3 Results

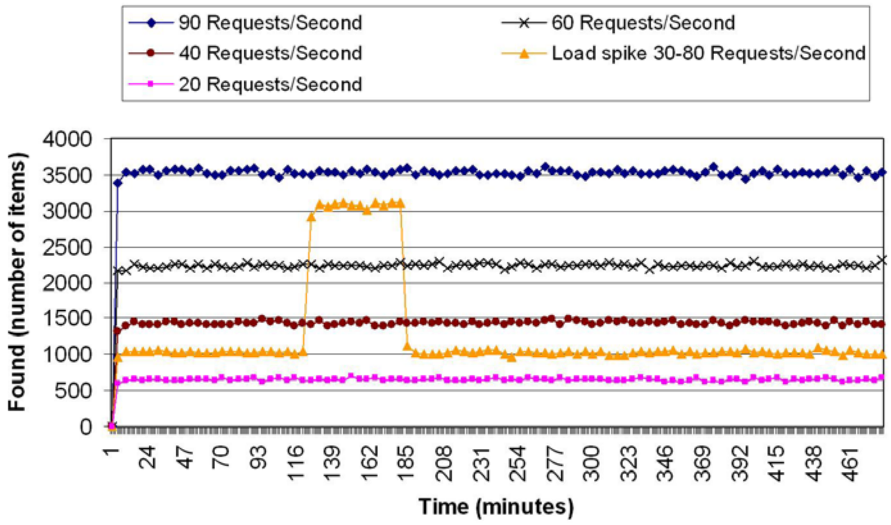

Figure 3: Number of items *found* over time for varied requests per second.

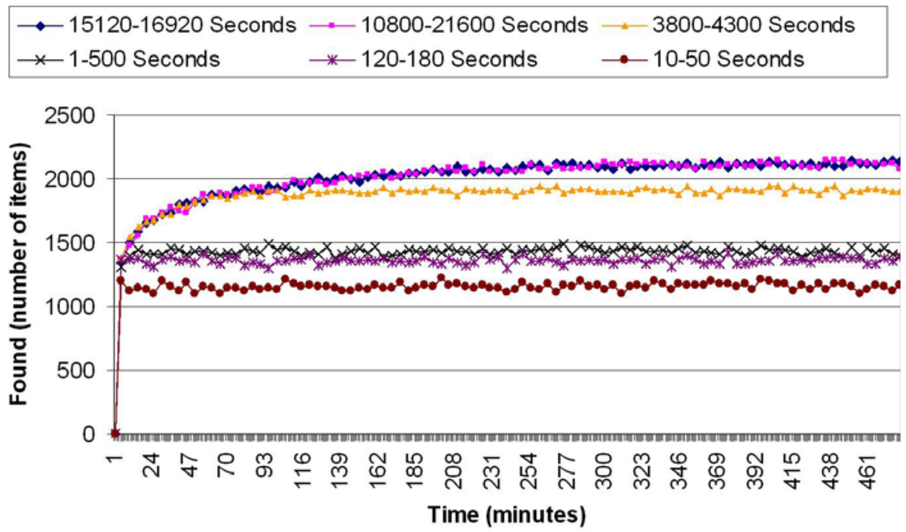

Figure 4: Number of items *found* over time for varied distribution times.

Figure 3 and Figure 4 illustrate how the number of items *found* changes over time for varied load and varied peer characteristics. In Figure 3, we fix the minimum and maximum distribution times at 1 and 500 seconds respectively. We observe that the greater the number of requests per second seen by the network, the greater the number *found* items at each 1 minute interval. This is not surprising since a greater number of requests will mean that the schedule of distributions is larger and there is a greater probability of overlap.

We also observe that, in all cases including the case when the load spikes from 30 to 80 requests per second from minute 120 to minute 180, the number of found items stabilizes quickly and remains stable throughout the experiment. This property allows us to conclude that under varying load conditions, the system will remain stable.

Another interesting observation is that the *percentage* of requests that are found remains relatively stable throughout the experiment. The percentage of found items ranges from 54.0% overall in the 20 requests per second case to 65.1% overall in the 90 requests per second case. Thus, we can extrapolate that even under varying load conditions, nearly the same percentage of requests will be found overall.

In Figure 4 we fix the load at 40 requests per second and vary the item distribution time. The item distribution time is the amount of time it takes to distribute a particular item. We observe that the greater the distribution time, the greater the number of *found* items at each one minute time interval. The reason for this behavior is that items with longer distribution times will remain in the schedule longer. Hence, the schedule itself will be larger and the probability of finding an item in the schedule will be higher. Additionally, when items have longer distribution times, the system takes longer to stabilize. This is because no items are removed from the schedule until the initially scheduled distributions finish.

We also observe that faster distribution times result in fewer found items overall. This is simply because when requests are processed faster, there is less opportunity to find a scheduled or executing distribution. Our results indicate that when downloads occur very quickly (10-50 seconds), the percentage of items found in the schedule is 48.1%. This is still a substantial percentage and would still render our system useful.

Our final observation is that slower connections with low variance tend to be quite cyclic. This is largely because the low variance means that all requests initially scheduled are likely to finish at nearly the same time and new requests will be scheduled at that time. This behavior is less likely to occur in a system with varying load, or one in which the load gradually builds up to a stable point.

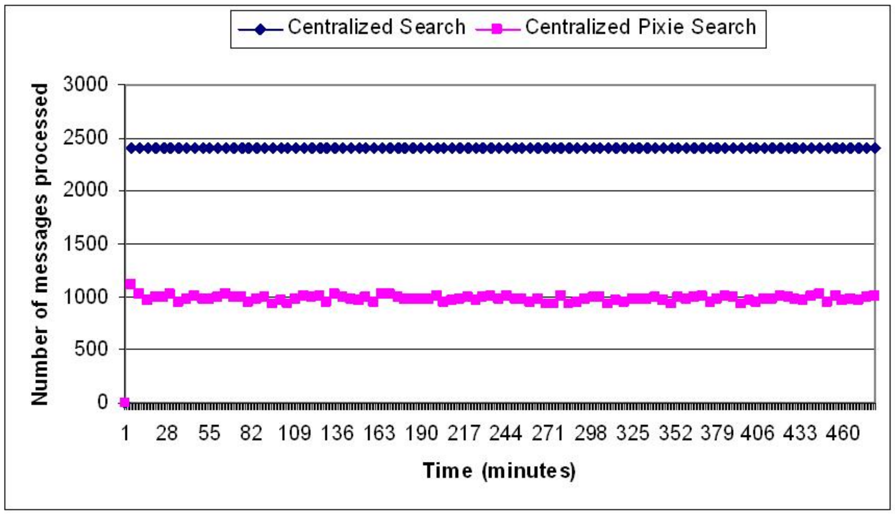

Figure 5: Number of search messages processed in a centralized scheme.

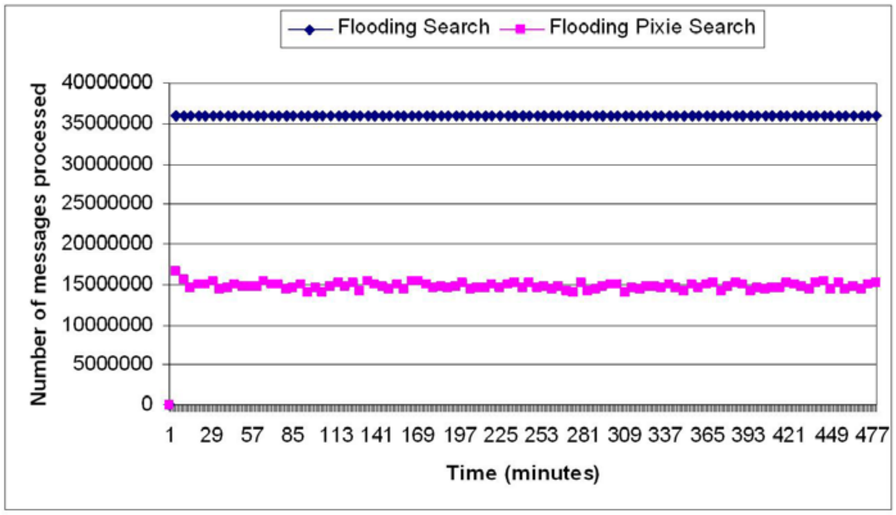

Figure 6: Number of search messages processed in a flooding scheme.

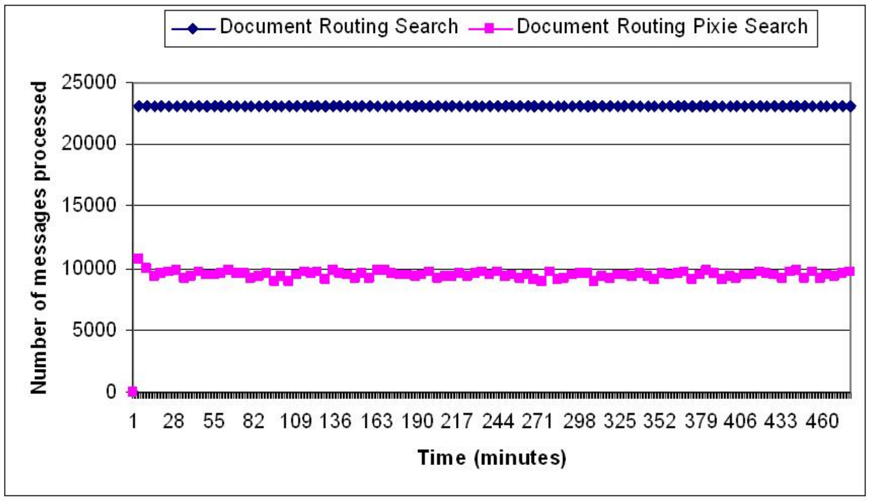

Figure 7: Number of search messages processed in a document routing scheme.

To quantify the search savings in *Pixie*, Figure 5, Figure 6, and Figure 7 illustrate the number of search messages that are processed in *Pixie* versus three standard search schemes. In each case, we assume that *Pixie* runs over the corresponding group management scheme we compare against. For example, in Figure 5, we assume that *Pixie* runs on top of a centralized group management protocol. In this case, we assume the existence of a Napster-like centralized server where all metadata is stored. When a *Pixie* peer becomes interested in a piece of content, it sends a search message to the centralized server which responds with a list of peers which have the desired content. To be fair, we compare this **Centralized *Pixie* Search** scheme to a Napster-like centralized search scheme which would incur the same additional overhead such as the initial upload of metadata. We also look at **Flooding *Pixie* Search** and **Document Routing *Pixie* Search** in comparison to a Gnutella-like scheme and a DHT-like scheme respectively. In the flooding case, we assume that *Pixie* runs on top of a Gnutella-style network. We assume that when a *Pixie* peer becomes interested in a particular piece of content, it sends a message to all of the peers it knows about. All of those peers send the message to all peers they know about, and so on. Our assumption is that the search message is processed by all peers in the network. In the document routing case, we assume that *Pixie* runs on top of a DHT such as CAN. In this case, we assume that the respective routing algorithm is used, and that *log N* peers process each search message (where *N* is the total number of peers in the network). We assume a network size of 15,000 nodes which is consistent with recent studies of the Gnutella network [14].

Our first observation is that *Pixie* reduces the total number of search messages processed by over half. In a centralized network, the savings is about 1,400 messages per minute. However, in a document routing network the savings is roughly 14,000 messages per minute, and in a flooding scheme, the savings is over 21,000,000 messages per minute. The disparity can be explained by the fact that each search scheme requires a different number of peers process each search message. Given our network size of 15,000 nodes, using *Pixie* instead of a flooding scheme saves approximately 24 messages per second per peer.

We also observe that the shapes of the curves for the three classes of search scheme are nearly identical. This is the result of using the same trace data for each experiment. However, as previously observed, the scale of the results varies for each scheme. A flooding scheme generates significantly more traffic overall than a centralized scheme. Therefore, *Pixie* is more beneficial in the flooding case.

Our final observation is that all schemes are very stable. The stability of the centralized, flooding, and document routing schemes can be explained by the fact that we assume that the network is stable, and we assume that the same number of messages are processed for each search. In reality, factors such as network instability, user behavior, and network configuration could affect the stability of these schemes. However, we also observe that *Pixie* is nearly as stable as the schemes we compare against. Therefore, in a stable network, *Pixie* is likely to be well behaved and consistently perform well.

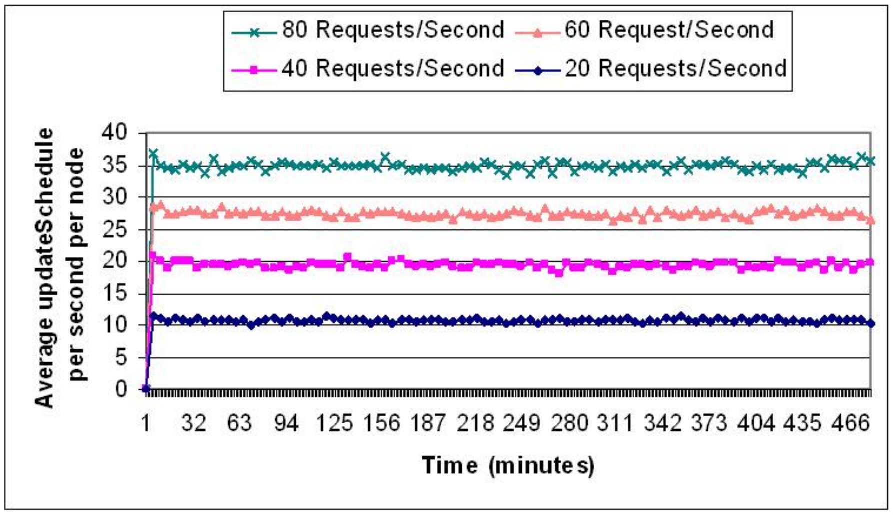

Figure 8: Number of updateSchedule messages processed at each node per minute.

While *Pixie* dramatically reduces the number of search messages processed in a P2P network, *Pixie* does incur the cost of additional updateSchedule messages that are flooded throughout the network for every scheduled distribution. In Figure 8, we examine the number of updateSchedule messages processed at each node in the network. We vary the load from 20 requests per second to 80 requests per second and look at the average number of updateSchedule messages per second processed at each node for every one minute time interval.

Our first observation is that the more heavily loaded the network, the more overhead incurred. This follows from the fact that a more heavily loaded network will have more scheduled distributions and hence more updates. However, we notice that even in a heavily loaded network supporting 80 requests per second, fewer than half the requests result in an updateSchedule message. Moreover, in a moderately loaded network of 40 requests per second, each node sees fewer than 20 updates per second. We believe this to be a reasonable tradeoff for the search savings and enhanced user experience gained by using the *Pixie* scheduling scheme.

We further observe that the number of updateSchedule messages sent remains stable throughout the 480 minute run of the experiment. Therefore, we can conclude that in a stable network, *Pixie* will remain stable as well.

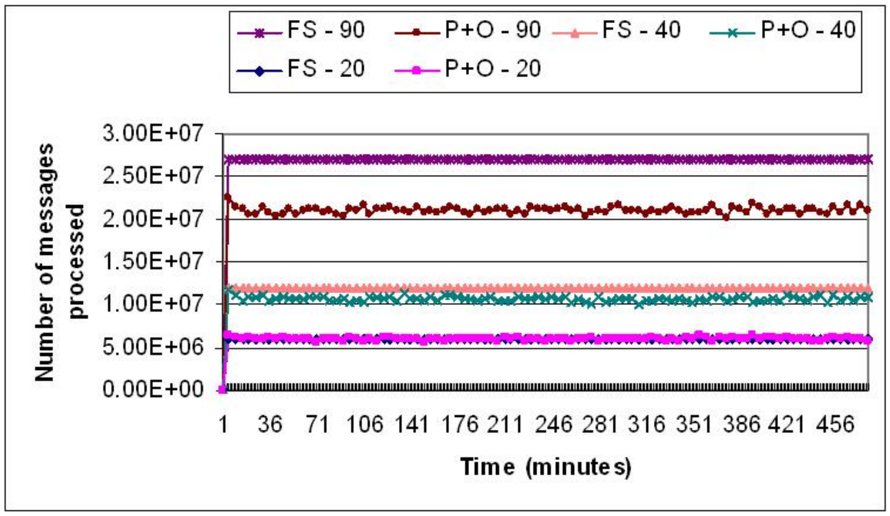

Figure 9: Search and search plus overhead in a small network.

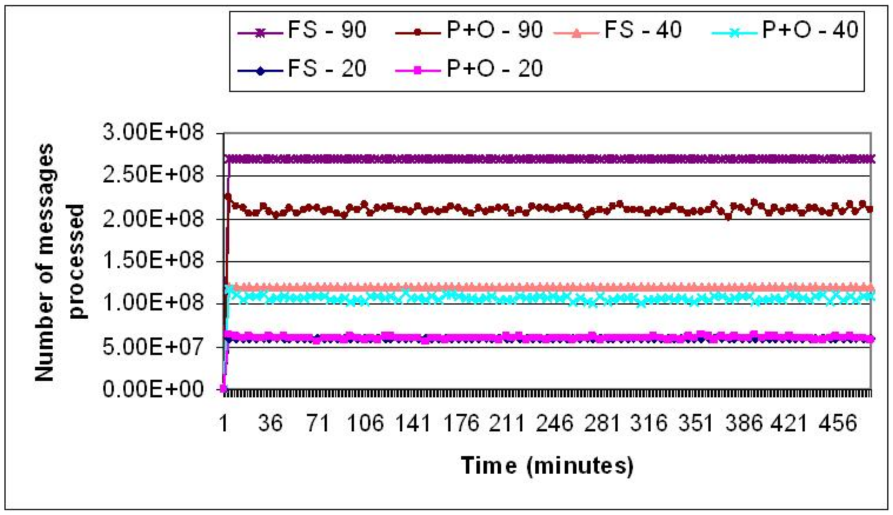

Figure 10: Search and search plus overhead in a large network.

Our final set of experiments further examines the tradeoff between the search savings gained by using *Pixie* and the overhead required by the updateSchedule messages. We present the number of search messages processed in a straightforward flooding search (FS) and the total number of search messages plus the number of updateSchedule messages (overhead) processed in a *Pixie* flooding search scheme (P+O). We vary the load from 20 requests per second to 90 requests per second and look at the total number of messages processed during each one minute time interval. In Figure 9 we assume a small network of 5,000 nodes and in Figure 10 we assume a large network of 50,000 nodes. We omit the results of the same experiment run on a medium-sized network of 15,000 nodes for clarity of presentation. However, the results were as expected.

Again, we use the same trace data for each experiment and thus observe the same phenomenon we observed in Figures 5, 6, and 7. Our curves are consistent, however, the larger the network, the more absolute savings. We see about a ten fold increase in savings from roughly 1,200,000 messages per minute to roughly 12,000,000 messages per minute from the 5,000 node network to the 50,000 node network.

We further observe that the more heavily loaded the network, the greater the disparity between *Pixie* and the flooding search scheme. Therefore, the more heavily loaded the network, the more benefit we gain by using *Pixie*. Even in a lightly loaded network, *Pixie* requires roughly an equivalent number of total messages to the flooding scheme. Further, we contend that updateSchedule messages require less processing and therefore, even in a lightly loaded network, we benefit from using *Pixie*.

4.4 Discussion

Pixie introduces a new model for P2P content exchange. From a user's perspective, *Pixie* means that P2P is no longer simply pull-based, on-demand information exchange. Information about the most popular content is actually *pushed* to the end user. Our results show that, in a moderately loaded system, over half of the time the user can find an item of interest by simply browsing the schedule of content. This not only improves the user experience, it greatly reduces the number of search messages flowing in the network.

However, there is a tradeoff between the search savings in *Pixie*, and the minimal overhead incurred by the updateSchedule messages. Fortunately, our results indicate that the updateSchedule message overhead is manageable. In fact, *Pixie* outperforms a flooding search network like Gnutella even with the additional updateSchedule overhead. Therefore, we conclude that *Pixie* provides both a desirable and efficient data location service.

5 Evaluation of *Pixie* Distribution

In this section, we quantify the benefit of aggregating requests and batching content distribution in P2P networks. Using *Pixie*, we can reduce the load on the serving peer as well as the wait time of the requesting peers.

5.1 Metrics

We are interested in two primary metrics:

1. **Wait Time** – In order to evaluate the benefit aggregation provides to the client, or requesting peer, we look at wait time. Wait time describes the amount of time the client must wait from the time it requests a piece of content until the distribution begins.
2. **Number Serviced per Distribution –** To evaluate the benefit aggregation provides to the serving peer, we look at the number of clients satisfied with each distribution. Using this metric, we can extrapolate on the resource savings of using an aggregation scheme.

5.2 Setup

To evaluate these metrics, we have simulated a single peer receiving and servicing requests. Since most peers act in a similar manner, modeling a single peer will provide us with adequate information to evaluate the desired metrics. If the peer receives a request for content that is already scheduled but has not begun, that request is aggregated and will be serviced by the already-scheduled distribution. If the request is for content that is not scheduled or is already being distributed, the peer schedules a new distribution of the requested content. We compare three scheduling schemes with respect to our target metrics.

We generate our traces using the parameters outlined in Section 4. Unless otherwise noted, experiments are run for 480 minutes, item distribution time is between 1 and 500 seconds, 40 requests per second are made across the entire network, and the serving peer stores one piece of moderately popular content. Of the 40 requests per second made across the network, only those requests for the content stored on the serving peer will be processed. All scheduling is done first come first served with respect to the requests for content. We discuss each of the scheduling schemes evaluated in more detail:

1. **FCFS –** This is the base case, *first come, first served*, no aggregation-no delay scheme. Distribution is one-to-one as is the case is current P2P systems and requests are serviced as soon as they are received.
2. **AGG-<DELAY> –** This is an *aggregation-delay* scheme. Multicast distributions of requested content are scheduled with delay <DELAY>, specified in minutes.
3. **DF-<DELAY>-<MAXDIST> –** This is a *digital fountain-delay-maximum distribution time* scheme. Digital fountain style distributions [20], are scheduled with delay <DELAY> as in the previous scheme. In addition, since the digital fountain scheme can cause starvation if requests for the same content continue to arrive, <MAXDIST> is a variable that specifies the maximum number of times a single distribution can be extended.

5.3 Results

We present the results of two experiments. In the first experiment we look at our target metrics in the base case. In the second experiment we take a slightly different look at the wait time metric while varying the load across the network.

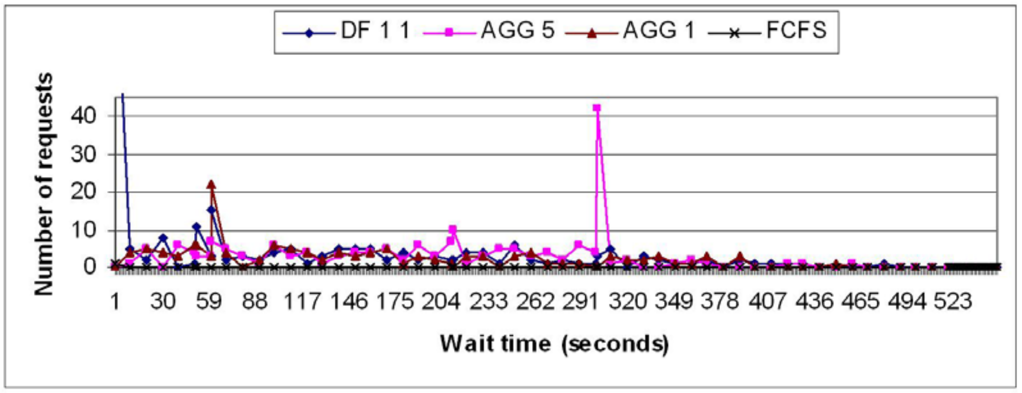

Figure 11: Number of requests experiencing each wait time at 40 requests per second.

Figure 11 plots the number of requests that experience each given wait time throughout the 480 minute experiment. In the FCFS case, many requests are made and queued, but not serviced within the 480 minute window. This is a common occurrence and illustrates the instability of the system using a FCFS scheme. Unserviced requests will remain in the queue of waiting requests at the end of the 480 minutes and are not reported here. We have truncated the FCFS data for presentation, but what happens in the FCFS case is that most of the serviced requests are issued in the first few timesteps. Because they are serviced sequentially, each request waits from the beginning of the experiment until the time it is serviced and the time waited increases linearly for each serviced request. In fact, the final request serviced waits for 25,487 seconds.

While the FCFS wait times increase linearly, in all aggregation schemes compared, the wait time remains relatively constant. Using aggregation, no request ever waits for greater than 500 seconds because, in the worst case, a request will be issued just as a distribution is starting, hence the request will have to wait the duration of the distribution. This worst case would be affected if the serving peer stored more than one piece of content. In the worst case, a peer storing *N* pieces of content would schedule them sequentially, *1, 2,...,N*. If a request for *1* was issued right after the distribution began, the request would have to wait $\sum_{i=1}^{N}$ *distribution_time*$_i$ seconds until *1* was scheduled again. A peer could potentially distribute multiple pieces of content simultaneously, but the time to complete each distribution is still restricted by the peer's outgoing bandwidth. Additionally, we suggest that by enabling users to browse a schedule of content, requests are likely to be influenced by content already scheduled.

Another observation of Figure 11 is that the spikes at wait times 0 (not shown for clarity of presentation), 61, and 301 indicate that the largest number of requests wait for the amount of time specified by the delay of the aggregation scheme. This is because the system is somewhat lightly loaded and relatively few requests arrive between the time that a distribution is scheduled and when it begins. While a longer delay implies a longer average wait time, the tradeoff is that a longer delay scheme utilizes fewer resources at the serving peer.

Finally, the difference between straightforward aggregation and a digital fountain-style aggregation scheme is minimal. This is primarily because we delay any new distributions by 1 minute and because we restrict the number of times the distribution can be extended to one. In fact, since our serving peer in this experiment stores only one piece of data, in an unrestricted digital fountain scenario we would achieve 0 wait time for all requests. If a peer was stable, likely to remain available, and stored only one or a few pieces of popular content, using an unrestricted digital fountain scheme would be the best choice.

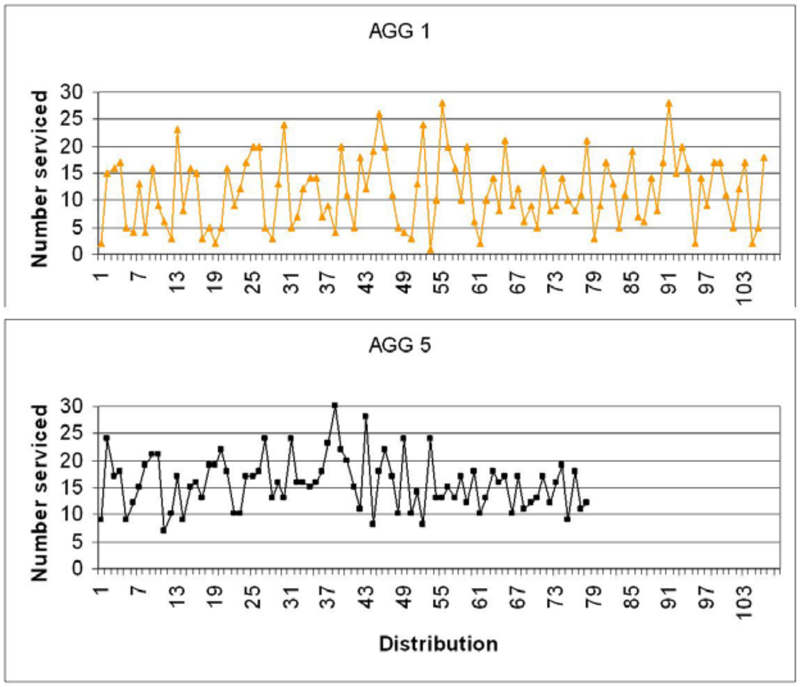

Figure 12: Number of requests serviced with each distribution at 40 requests per second.

Figure 12 illustrates the number of peers serviced for each distribution scheduled during the 480 minute run using the same parameters used for the experiment shown in Figure 11. We omit the results of the DF 1 1 scheme for presentation since the results were similar to the AGG 1 scheme. We also omit the results of FCFS because it always services one request.

We notice that the aggregation schemes manage to service up to 30 requests per distribution. We also notice that the AGG 5 scheme never services less than 7 requests per stream while the AGG 1 scheme often services fewer. Because AGG 5 consistently services more requests than the lower delay scheme, fewer distributions are required. This is simply because more requests are aggregated prior to the beginning of a given distribution. What this implies is that there is a tradeoff between the resources used at the serving peer and the wait time experienced by the client. By incurring an average wait time penalty of 33 seconds with the AGG 5 scheme, we gain roughly a 25% resource savings at the serving peer. Our final observation is that by using aggregation, we gain an advantage in terms of disk space required across the network. For the FCFS scheme to achieve the same performance of the AGG 5 scheme, content would have to be replicated up to 30 times throughout the network.

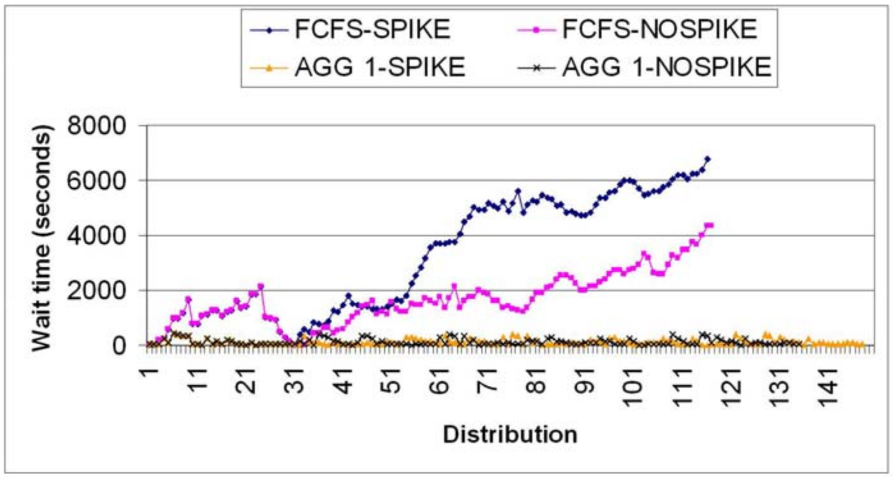

Figure 13: Wait time for each request serviced with load surge.

In Figure 13 we store an unpopular piece of content on the serving peer and demonstrate a spike in load from 40 to 90 requests per second. The figure illustrates the wait time experienced under these conditions. While wait time with the FCFS scheme continues to grow, even without the load spike, wait time using the aggregation scheme remains stable over all requests. Such behavior is especially important when a new, popular piece of content is introduced into the peer network. In the best case FCFS scheme, distributing a single piece of content throughout the entire network would be logarithmic with respect to the number of peers. Using aggregation, the same distribution occurs in constant time.

5.4 Discussion

The one-to-many nature of *Pixie* distribution provides a number of additional scalability properties. *Pixie* dramatically reduces the wait time experienced by client peers while reducing the load on the serving peer. A peer in a standard, first come, first served scheme can wait over 50 times longer between the request for content, and the start of a distribution than a *Pixie* peer. Additionally, the serving peer load can be reduced by up to 30 times by servicing multiple peers in parallel. Finally, *Pixie* is virtually unaffected by an increase in load across the peer network. It gracefully handles heavy load as well as load spikes whereas the first come, first served case becomes very unstable in both situations. Previous experiments [3] have also shown that *Pixie* performs well under varied network load and peer characteristics as well.

6 Reliability and Fault Handling

The dynamic nature of peer networks makes reliable and fault tolerant peer-based applications particularly challenging to implement. A peer that is distributing or simply routing content may go offline at any time. In this section, we outline how *Pixie* tolerates serving peer faults and discuss the performance impacts of our fault handling scheme.

6.1 Serving Peer Fault Handling

By using a digital fountain scheme or straightforward reliable multicast scheme for reliability either over a native or application-layer multicast infrastructure we can ensure that data lost between the serving peer and the receiving peer are eventually recovered. However, we have not yet addressed how we recover from serving peer faults. The most critical failure case occurs when a serving peer fails during or before its scheduled distribution. Failure can be the result of system failure, network failure, or a user can simply choose to take the peer offline, the so-called *freeriders* problem. We make the assumption that peers may go offline without any prior notification. Therefore, we must develop a strategy for rescheduling distributions that have not completed.

First, let us consider the basic (e.g., no failure) case when a user does not find an item of interest in the schedule and must search for a piece of content and schedule a distribution.

1. An interested peer searches for a piece of content.
2. Once the piece of content has been found, the requesting peer contacts the serving peer and requests a distribution be scheduled.

3. An updateSchedule message is propagated to all peers in the network.
4. At distribution time, the content is distributed to all interested peers.

In the event that the serving peer fails, there must be a strategy for detecting the failure and rescheduling the distribution. For the purposes of this work, we assume that a serving peer has failed if a receiving peer expects to receive content from the serving peer, but has not received data for some *timeout* period. The determination of this timeout period is dependent on many factors, including the underlying data distribution protocol. Moreover, more advanced schemes for detecting failures are also possible. However, more sophisticated schemes are beyond the scope of this work.

Once a fault is detected, the distribution must be rescheduled. To support our fault recovery algorithm, we must modify the basic search and schedule case as follows:

1. An interested peer searches for a piece of content.
2. Once the piece of content has been found, the requesting peer contacts the serving peer and requests a distribution be scheduled.
3. Along with the request, the requesting peer provides a list of all potential serving peers in the network that initially responded to the search request.
4. An updateSchedule message that includes the complete list of potential serving peers is propagated to all peers in the network.
5. At distribution time, the content is distributed to all interested peers.

Using the cached search results, the general fault recovery algorithm is as follows:

```
if the serving peer fails
  wait a random backoff period
  if the distribution has not resumed
    do
      select an alternate serving peer from the cached
          search results
    while the selected peer is not reachable
    request a new distribution from the selected peer
          on the same channel
    immediately, the new serving peer begins
          distribution to all peers waiting for the
          distribution
```

This strategy reschedules a failed distribution in an efficient manner. First, the random backoff period helps to avoid the case that multiple receiving peers simultaneously detect a fault and attempt to reschedule a distribution. The first peer to reach the end of the backoff period will reschedule the distribution and the remaining peers can take advantage of the rescheduled distribution. Moreover, using the same channel (e.g., multicast group address) for the rescheduled distribution avoids the overhead of sending out an additional updateSchedule message. Only those peers that are joined in the current distribution are affected. Finally, because the original search responses are cached in the schedule, there is no need to burden the network with a new search.

More advanced and efficient schemes for handling faults and ensuring reliability are also possible. For example, assuming the use of a digital fountain-style scheme, if multiple peers store the same content, they can be scheduled to distribute the same file simultaneously on the same channel. If neither serving peer fails, those receiving peers with a high bandwidth connection can potentially receive the file in half the time. Also, if one serving peer fails, another peer is already in the process of distributing the content. Moreover, if we used a more reliable group management protocol such as Chord, we could detect the failure of a peer and automatically reschedule a distribution at the infrastructure layer.

6.2 Performance Analysis

Our serving peer fault handling scheme incurs minimal cost. The goal of this paper is to compare *Pixie* against a typical P2P file sharing application such as Gnutella or Napster. Both of these applications leave fault tolerance up to the user by requiring that the user resubmit search queries or download requests in the case of failure. Comparing *Pixie*'s fault tolerance mechanisms to these typical fault tolerance mechanisms would be an unfair comparison. In this section, we discuss the comparison between *Pixie* in a faulty network and *Pixie* in a stable network. However, it is worth noting that many of the performance penalties discussed also apply to most if not all of the P2P file sharing systems which have been deployed to date. Those that do not are penalties associated with metrics unique to our system.

First, the metrics we evaluate in Section 4 and Section 5 are not directly affected by serving peer failure. **Found** remains the same even if peers fail because the schedule is kept locally. Though, it is possible that a peer has stale information in its schedule. In other words, a peer, *p1*, could have in its schedule an entry which denotes that another peer, *p2*, has a piece of content even if *p2* has failed. The penalty is simply that *p1* may have to reissue a schedule request if it makes the initial request to a failed peer.

Number of search messages processed is also likely to remain the same even if peers fail. The only time it would increase with increased peer failure would be when *all* serving peers associated with a particular piece of content in the schedule failed between the time the corresponding updateSchedule message was applied to the schedule and when it was removed.

The **updateSchedule Message overhead** would be modestly larger because our fault tolerance scheme requires that the updateSchedule message contain an entry for all peers that have the advertised piece of content. This will add a few Kilobytes to the updateSchedule message size as well as to the size of the schedule. However, we do not believe this will impact the performance of the system.

Wait time will be unaffected unless a serving peer fails before it even begins to deliver content. In this case, there will be an additional delay associated with contacting an alternate serving peer and rescheduling the distribution. Because the schedule has information about all potential serving peers, the delay will not include any search time.

Number serviced per distribution will only improve in a faulty network. If there are fewer serving peers to service requests, more requests will be aggregated and serviced simultaneously using our one-to-many delivery scheme.

In addition, if we employ the proposed digital fountain style scheme for reliability, there is a small penalty associated with using Tornado codes. Some redundant information is introduced and the impact is that the number of bytes a receiving peer must receive in order to reconstruct the original file is 5-10% greater [20] if we assume no serving peer failures. However, the benefit of using Tornado codes is that if a serving peer does fail, a receiving peer can make use of the content it has already received rather than starting again from the beginning. If most failures occur after a serving peer has delivered at least 10% of a data file, our scheme will outperform typical P2P content delivery.

Finally, there is a delay associated with rescheduling an in-progress distribution. If a serving peer fails, one of the receiving peers must request a new distribution from another serving peer with the same content and wait for that serving peer to begin delivery. However, because each peer has a cache of information about where content is available, the receiving peer which requests the new distribution does not need to perform another search of the network. Additionally, unlike typical P2P file sharing systems, we do not rely on the user to initiate the new delivery.

7 Concluding Remarks

Pixie is an architecture designed to improve data location and content distribution in peer-to-peer networks. In *Pixie*, peer requests for content are aggregated. A schedule of distribution times is propagated throughout the network, and distribution is done using a one-to-many distribution scheme. Thus, a virtually limitless number of peers may take advantage of the same distribution.

In this work, we expand our previous work in two ways. First, we examine *Pixie*'s integration of well-known reliable distribution protocols and propose an algorithm for handling serving peer faults. Second, we evaluate the tradeoff between the overhead associated with maintaining the schedule and the search savings gained by using the schedule abstraction. *Pixie* can eliminate search overhead by nearly 60% and, even with the schedule maintenance overhead, outperforms a straightforward flooding search scheme. Additionally, we summarize our previous results which indicate that *Pixie* can greatly reduce client wait time and server resource usage.

Pixie also introduces a number of interesting areas for future research. First, we would like to integrate many-to-many content distribution into the *Pixie* architecture. However, there are still a number of unsolved challenges associated with many-to-many content distribution such as which potential servers should be selected to distribute content. We are also interested in evaluating *Pixie*'s behavior in an unreliable network. We anticipate that, using *Pixie*, it is likely that more peers will ultimately be serviced under unreliable conditions.

Peer content exchange is becoming more popular and encompasses an increasing number of applications. The content being exchanged is becoming larger while the peer devices are becoming smaller. One-to-one distribution in these scenarios is inefficient if not impossible. Moreover, locating content in larger and more diverse networks is even more of a challenge. Using *Pixie*, we can reduce the impact of these challenges and make content exchange more efficient.

References

[1] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrisnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Sigcomm 2001*, (San Diego, CA, USA), Aug. 2001.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Sigcomm 2001*, (San Diego, CA, USA), Aug. 2001.

[3] S. Rollins and K. Almeroth, "*Pixie*: A jukebox architecture to support efficient peer content exchange," in *ACM Multimedia*, (Juan Les Pins, France), Dec. 2002.

[4] M. Neary, S. Brydon, P. Kmiec, S. Rollins, and P. Cappello, "Javelin++: Scalability issues in global computing," *Concurrency: Practice and Experience*, vol. 12, pp. 727–753, 2000.

[5] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, "Peer-to-peer computing," Tech. Rep. HPL-2002-57, Hewlett Packard Laboratories, 2002.

[6] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination," in *NOSSDAV*, (Port Jefferson, NY, USA), June 2001.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, (Berkeley, CA, USA), July 2000.

[8] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location an d routing," Tech. Rep. UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

[9] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Middeware*, (Heidelberg, Germany), Nov. 2001.

[10] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP 2001*, (Banff, Canada), Oct. 2001.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon,W. Weimer, C.Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *ASPLOS*, (Cambridge, MA, USA), Nov. 2000.

[12] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent, peer-to-peer storage utility," in *SOSP 2001*, (Canada), Nov. 2001. [13] E. Adar and B. Huberman, "Free riding on gnutella," *First Monday*, vol. 5, Oct. 2000.

[14] S. Saroiu, P. Gummadi, and S. Gribble, "A measurement study of peer-to-peer file sharing systems," in *MMCN*, (San Jose, CA, USA), Jan. 2002.

[15] S. Rollins, R. Chalmers, J. Blanquer, and K. Almeroth, "The active information system (AIS):A model for developing scalable web services," in *Internet Multimedia Systems and Applications*, (Kauai, Hawaii, USA), Aug. 2002.

[16] K. Almeroth and M. Ammar, "The interactive multimedia jukebox (IMJ): A new paradigm for the on-demand delivery of audio/video," in *WWW7*, (Brisbane, Australia), Apr. 1998.

[17] S. Sheu, K. Hua, and T. Hu, "Virtual batching: A new scheduling technique for video-on-demand servers," in *DASFAA*, (Melbourne, Australia), pp. 481–490, Apr. 1997.

[18] D. Helder and S. Jamin, "End-host multicast communication using switch-tree protocols," in *Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, (Berlin, Germany), May 2002.

[19] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Sigcomm 2002*, (Pittsburgh, PA, USA), Aug. 2002.

[20] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Sigcomm*, (Vancouver, British Columbia), pp. 56–67, Sept. 1998.

[21] G. Zipf, *Human Behavior and the Principle of Least Effort*. Reading, MA: Addison-Wesley, 1949.

[22] M. Ripeanu, I. Foster, and A. Iamnitchi, "Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design," *IEEE Internet Computing Journal, Special Issue on Peer-to-Peer Networking*, vol. 6, no. 1, 2002.