A PROBABILISTIC APPROACH TO BUILDING LARGE SCALE FEDERATED SYSTEMS

BY FRANCISCO M. CUENCA ACUNA

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Thu D. Nguyen

and approved by

________________________________________

________________________________________

________________________________________

________________________________________

New Brunswick, New Jersey
October, 2004
ABSTRACT OF THE DISSERTATION

A Probabilistic Approach to Building Large Scale Federated Systems

by Francisco M. Cuenca Acuna
Dissertation Director: Thu D. Nguyen

Rising Internet connectivity and emerging web service standards are enabling a new federated computing model, where computing systems will be comprised of multiple components distributed across multiple collaborative organizations. While federated services can revolutionize collaboration and commerce across the Internet, the realization of this vision faces a number of challenges arising from its fundamental cross-organizational nature. In addition, problems like faulty hardware and software, operator error, dynamic reallocation of resources, load spikes, and malicious users cause these systems to be highly volatile making federated computing even more challenging.

Traditional techniques for building distributed systems, have generally provided resource management and communication by relying on structured solutions like: (a) imposing an overlay structure over the system (i.e. multicast tree or distributed hash table), (b) depending on centralized services or (c) relying on distributed consistency protocols. Previous work [42,46,64,102], has shown that these techniques become expensive and sometime unfeasible in environments where membership changes rapidly and nodes are heterogeneous and unpredictable.

In this dissertation, we explore a different approach for building large scale distributed systems. Our thesis is to create distributed algorithms that allow members to
operate autonomously so that their progress is not conditioned by other nodes. Despite their independence, as a whole members should be able to make constant progress toward achieving stable global goals. In order to ensure autonomy, global progress and stability, we build randomized algorithms that depend only on loosely synchronized global data.

In this dissertation we explore an infrastructure called PlanetP that we have simulated and partially prototyped to validate our thesis. PlanetP embodies several of our ideas by using probabilistic algorithms to provide, group communication, membership management, content based search and ranking, autonomous service deployment and management, and autonomous replication to provide predictable data availability. Our work is novel in that we target highly dynamic environments where nodes join and leave constantly in an uncontrolled manner.
Acknowledgements

Throughout these years I have had the pleasure of working, learning, having fun and suffering with several people here at Rutgers. All of them have contributed in different ways to my success and share a part of it.

Thu Nguyen, my advisor, has been a great person who has always pushed me to do “good research” while keeping a healthy life. Thanks Thu!.

Thanks also to the “lab people” (Christopher Peery, Kiran Nagaraja, Xiaoyan Li, Sudeept Bhatnagar, Taliver Heath, Eduardo Pinheiro, Ricardo Bianchini and Richard Martin) for all your support. In particular thanks to Chris, Kiran, Xiaoyan and Sudeept for sharing with me your views of this crazy world we live in and making this experience much more interesting.

Thanks to the “Argentinean gang” (Miguel & Silvia, Carlos & Valeria, Rodolfo & Adolfina, Roberto & Jessica, Christian and Marcelo).

Finally, thanks to the secretaries, LCSR and “the other profs” at Rutgers that helped me get through this experience.
Dedication

To Cecilia; who has chosen the tough job of being my life partner.

To my family; my roots, my essence.
Table of Contents

Abstract ................................................................. ii
Acknowledgements ....................................................... iv
Dedication ............................................................... v

1. Introduction .......................................................... 1
   1.1. Thesis structure .................................................. 3
   1.2. Contributions .................................................... 8

2. Related Work .......................................................... 10
   2.1. Distributed systems and Internet computing ................ 10
   2.2. Global scale infrastructural services ...................... 11
   2.3. Autonomous Replication ..................................... 13
   2.4. Content search and ranking ................................. 14
   2.5. Federated Services Management ............................ 15

3. Infrastructure ......................................................... 17
   3.1. Overview ......................................................... 17
   3.2. Gossiping ......................................................... 18
   3.3. The Global Directory ......................................... 20
   3.4. Performance ..................................................... 21
       3.4.1. Storage Cost ................................................ 23
       3.4.2. Propagating new information ............................ 24
       3.4.3. Joining of new members ................................. 28
       3.4.4. Dynamic operation ....................................... 30
   3.5. Summary .......................................................... 32
4. Data availability .................................................. 34
  4.1. Overview ................................................. 34
  4.2. Autonomous Replication ................................. 35
    4.2.1. Randomized Fragmentation and Replication .......... 37
    4.2.2. Estimating the Availability of Files ................ 38
    4.2.3. Replacement ....................................... 39
    4.2.4. Optimizations at Replicators ....................... 41
  4.3. Evaluating the Achievable Availability ................. 41
    4.3.1. Experimental Environment ........................... 41
    4.3.2. Simulated Communities .............................. 42
    4.3.3. Other Replication Algorithms ....................... 46
    4.3.4. Availability Improvements ......................... 46
  4.4. Summary .................................................. 53

5. Content search and ranking ................................. 54
  5.1. Overview ................................................ 54
  5.2. Background: TFxIDF ................................... 55
  5.3. Approximating TFxIDF .................................. 56
  5.4. Performance ............................................ 58
    5.4.1. Search Efficacy .................................. 59
    5.4.2. Replication & Content Ranking ...................... 64
  5.5. Summary ................................................ 67

6. Self managed federated services ............................. 68
  6.1. Overview ................................................ 68
  6.2. Self Management Approach and Algorithms ................ 69
    6.2.1. An Example Application Model ...................... 71
    6.2.2. Finding the Right Configuration .................... 74
    6.2.3. Realization of a Configuration ...................... 76
  6.3. Architecture ............................................ 78
6.3.1. Runtime Environments ........................................... 79
6.3.2. Node Monitoring ................................................... 80
6.3.3. Application Deployment and Management ...................... 80
6.3.4. The UDDI Service .................................................. 81
6.3.5. The UDDI Clients .................................................. 81
6.4. Evaluation ............................................................. 82
6.4.1. Experimental Environment ...................................... 82
6.4.2. Behavior on Cluster Testbed ..................................... 84
6.4.3. Behavior on PlanetLab ............................................ 87
6.5. Summary ............................................................... 89

7. Applications ............................................................. 90
7.1. Wayfinder ............................................................. 90
7.2. Vivo ................................................................. 92

8. Conclusions and Future Work ........................................ 93
8.1. Future Work .......................................................... 95

References ................................................................. 98
Vita ................................................................. 107
Chapter 1

Introduction

Rising Internet connectivity and emerging web service standards are enabling a new federated computing model, where computing systems will be comprised of multiple components distributed across multiple collaborative organizations. The emergence of federated computing is already evident at several levels of collaboration, including peer-to-peer (P2P) information sharing [37,59], scientific computing grids [28], and e-commerce services [3]. As an example, the European Data Grid [28] is a consortium aggregated across several countries that share computing infrastructure, scientific instruments and experimental data. Similarly, PlanetLab [77] is a global effort to create a shared computing infrastructure that can be used by researchers to evaluate new technologies for building distributed systems.

While federated services can revolutionize collaboration and commerce across the Internet, the realization of this vision faces a number of challenges arising from its fundamental cross-organizational nature. Federated systems pose new challenges that limit the efficacy of traditional techniques for building distributed systems. For example, classic distributed operating systems like Sprite [71], Amoeba [68], and Locus [78] depend on centralized services like transaction coordinators and storage managers. This dependence on centralized resources is not well suited for environments like the European Data Grid where countries may have different policies that cannot be embodied within a single shared centralized resource. For example, countries may disagree on the maintenance of a server depending on how it affects their own use of the infrastructure.

Not only political reasons, but also technical limitations prevent the use of existing systems. A possible solution to decentralize and distribute the control of federated environments is to replicate critical services across organizations. Unfortunately traditional
consistency protocols like two phase commit, leader election and virtual synchrony have been found to scale poorly on wide area networks [42,46,102]. Gupta et al. [46] have shown that it only takes a single faulty or slow member to severely affect the performance of the whole community. They have also observed that this kind of problems becomes the norm when dealing with even small communities comprised of tens of peers distributed across a wide area network. Furthermore, Gray et al. [42] studied the problem of scaling replicated databases. Their results conclude that transactional replication is not feasible for large systems, as a ten fold increase on the number of replicas and clients causes a thousand fold increase in the number of deadlocks or reconciliations.

In this dissertation, we explore a different approach to building large scale distributed systems. Our thesis is to use distributed algorithms that allow members to operate autonomously so that their progress cannot be impeded by other nodes. Despite their independence, as a whole, members should be able to make constant progress toward achieving stable global goals. In order to ensure autonomy, global progress and stability, we build randomized algorithms that depend only on loosely synchronized (i.e. weakly consistent) global data.

In the past, probabilistic and randomized algorithms have been very successful at providing reliable and scalable solutions for problems like membership management [47], multicast communication [9], database replication [22,42], information aggregation [100], failure detection [101] and security [109]. Motivated by this growing trend in software design, we set ourselves to build a new distributed infrastructure to support federated services based on probabilistic distributed algorithms.

Our work is novel in that we target highly dynamic environments where nodes join and leave constantly in an uncontrolled manner. Further, problems like faulty hardware and software, operator error, dynamic reallocation of resources, load spikes, and malicious users cause these systems to be highly volatile. For example, Saroiu et al. [88] report an average node availability of only 24% on communities like Gnutella [37].
1.1 Thesis structure

In this dissertation we explore an infrastructure called PlanetP that we have simulated and partially prototyped to validate our thesis. Our discussion begins with the construction of a communication infrastructure, which we later use to loosely synchronize global data. Next, we consider two problems that appear on federated systems: (a) the location and retrieval of data in highly dynamic communities and (b) the management and monitoring of federated services.

Every component of PlanetP has been simulated to estimate its performance and scalability on communities of thousands of nodes. Throughout this dissertation we use simulated communities to study different aspects of real systems like the European Data Grid and PlanetLab. These communities are based on our own observations of real systems [24, 77] and previous work on the field [11, 29, 44, 48, 88, 106]. Furthermore, PlanetP has been successfully used to deploy federated services in a community composed of 100 PlanetLab [77] nodes across the US.

**Group communication and state propagation.** A significant amount of work has been done to support group communication [5, 8, 9, 13, 27, 55]. Still, as previous studies show, even models with weak delivery guarantees suffer significantly when used on volatile and unpredictable environments [46]. Thus, in Chapter 3, we propose a novel approach to the construction of a content addressable publish/subscribe service that uses Demers et al.’s [22] anti-entropy [22] algorithm to replicate global state across unstructured communities of several thousand nodes.

Briefly, Demers et al.’s algorithm for synchronizing a global data structure and propagating updates (i.e. broadcasting) works as follows: suppose $x$ has a piece of information of interest to the entire community. Periodically, $x$ would push this change (called a rumor) to a randomly chosen peer $y$. If $y$ has not seen this rumor, it records the change and starts to push the rumor just like $x$. In this manner, information eventually diffuses throughout the whole community.

We argue that gossiping is simple, yet it provides a powerful tool for sharing information. Gossiping is simple because each peer must only agree to perform a periodic,
randomized, point-to-point message exchange with other peers, rather than collaborate to correctly and consistently maintain a complex distributed data structure. Gossiping is powerful for two reasons: (a) it can propagate information in probabilistically bounded time in spite of the uncoordinated communal behavior, and (b) it can maintain loosely consistent replicated state without depending on centralized resources or the on-line presence of specific peers.

The latter allows PlanetP to maintain loosely synchronized communal databases. For example, PlanetP includes a default membership database, called the global directory, which allows federated services to get information about the community and the status of its members. As shall be seen, the global directory is also used to provide a multidimensional keyword index. This global index stores key-to-node mappings “k → n”, which are used to advertise that node n has a some particular content or resource (i.e. an object) associated with keyword k. Objects are then retrieved by specifying queries comprised of sets of keys combined using three operators, and (∧), or (∨), and without (¬). For example, a query (“cat” ∧ “dog” − “bird”) would contact all the nodes advertising the keywords “cat” and “dog” but not “bird” and then retrieve all the related objects.

Finally, using the global index PlanetP also provides a publish/subscribe abstraction where applications can register call-backs to be informed when new mappings that match a particular keyword k are published.

Data availability. Given the above infrastructure, we proceed to consider the problem of reliably sharing data in federated environments. One of earliest widespread use of federated computing was file sharing applications like Gnutella [37] and Freenet [19]. In these environments, files were manually replicated by users based on their popularity. Moreover, searching consisted of sampling a subset of the online community. As a result, uncontrolled file availability became a limitation for both search and retrieval. If we were to move file sharing beyond just music content into sharing scientific results or legal case studies, today’s pattern of availability determined by file popularity may not hold. Moreover, locating unpopular references may be critical to answering specific queries. Similarly, data availability can greatly influence the performance of more structured
approaches like federated file systems [61, 69] as they critically depend on key shared data like for example the root directory.

Recent measurements suggest that highly dynamic communities, like P2P groups, are fundamentally different from current servers [7, 88]; for example, Saroiu et al. report an average node availability of only 24% [88]. Such low availability implies that providing practical availability for shared data, say 99-99.9%, which is comparable to today’s web services [65], would be prohibitively expensive storage-wise using traditional replication methods. Yet, requiring that data be moved or re-replicated as nodes leave and rejoin the online community would be prohibitively expensive bandwidth-wise. For example, replicating a file 6 times (i.e. 1 original copy plus 6 replicas) when the average node availability is only 24% would only raise its availability to 85%. Moreover, if a thousand nodes were to share 100GB (700GB after replication), the bandwidth required to keep all replicas online as members join and leave the online community would be around 4GB per node per day (see Chapter 4).

We are thus motivated to study how to improve the availability of shared data for highly dynamic communities where individuals may be disconnected as often as being online. In particular, assuming that nodes eventually rejoin the online community when they go offline, we address the question: is it possible to place replicas of shared files in such a way that, despite constant changes to the online membership, files are highly available without requiring the continual movement of replicas? To answer this question, we propose and evaluate a distributed replication algorithm where all replication decisions are made autonomously by individual members using only a small amount of loosely synchronized global state shared through PlanetP’s global directory. To maximize the utility of excess storage, we assume that files are replicated in their entirety only when a member hoards that file for disconnected operation. Otherwise, files are replicated using an erasure code [104]. While the use of an erasure code is not novel, we show how to avoid the need for tracking the placement of specific fragments. In particular, we show how to increase the availability of a file by adding new random fragments rather than regenerating individual fragments that have been lost.

Content search and ranking. As communities get larger, locating relevant data
becomes an important issue \[18, 49, 63, 95\], consequently we proceed to study how to provide content search and ranking on federated environments. The success of Internet search engines in indexing newsgroups and mailing lists (e.g., Google Groups) as well as the web in general argues that content-based search and ranking is a powerful model for locating information across data collections exhibiting a wide range of sizes and content.

Our studies of a local P2P community comprised of 3000 University students, show that they currently share over 20TB of information. Similarly, we find that scientific grids aggregate petabytes of data with relatively few nodes (e.g. 160 nodes in the case of the European Data Grid). Locating data in these large federated environments is challenging because shared information is not centralized and as the community membership changes, so does the content that is available for searching. Traditional information retrieval techniques that rely on crawling to build a centralized index, would need to spend lots of resources polling for updates to be able to keep up with this dynamicity. On Chapter 5, we propose a scheme for searching and ranking documents distributively without building such index. This scheme is based on the popular vector space ranking model \[107\] used on Internet search engines like Altavista.

Further, using the autonomous replication algorithms presented on Chapter 4, we study how to provide high availability for indexing metadata. Effectively, we address the issue of how to being able to search, rank and download a file with predictable availability. Figure 1.1 summarizes the relation between the different layers that we have described so far.

**Self managed federated services.** Now we turn to addressing the sharing of computing resources. In particular, we focus on building self-managing federated services. Recent studies \[73\] show that operator errors are typically the largest cause of Internet service failures. Oppenheimer et al. \[72\] show that 19% to 33% of the total errors are caused by the operators and more than 50% of these are due to faulty configurations.

In federated environments, deploying and managing services becomes an even more difficult task. In order to realize the federated computing promise, we must first raise the abstraction level at which systems operators currently manage federated services. In particular, applications must be able to autonomously execute in heterogeneous and
Figure 1.1: PlanetP’s architecture: going bottom up we depict all the layers that deal with information management, i.e. the communication layer used to build a global index, the content search and ranking layer, and the storage layer used to provide predictable data availability.

potentially highly dynamic environments that are not under the control or management of any single entity.

Following our goal of building infrastructural support for federated environments, we study the case of critical services that can cause the failure of entire systems. For example, the European Data Grid relies on a continental scale directory service comprised of more than 100 data providers. This directory service is vital to the operation of the grid, yet its topology and redundancy must be manually configured. Any adjustments or changes must also be performed manually.

Similarly, consider services such as the infrastructural Universal Description, Discovery and Integration (UDDI) [97] service that is used in federated environments to locate and access Web Services across organizations. Such multi-site services must currently be configured and maintained by hand, placing a considerable burden on system operators [108]. Worse, it is quite difficult to understand and properly manage the overall
performance and availability of such a service because the components are spread across multiple administrative domains.

In Chapter 6, we design a distributed resource management framework that can be used to build self-managing multi-site services that dynamically adapt to changing system configuration and client load. Our goal is to reduce the burden of deploying and managing a multi-site service to three tasks: (1) defining an application model for the framework to choose appropriate runtime configurations and specifying the desired availability and performance goals, (2) deciding the set of machines that can host instances of the service, and (3) providing maintenance support to repair these machines when they fail. Given a set of hosting machines, our framework will start an appropriate number of service replicas to meet the desired quality of service. Further, it will monitor the application as well as the hosting infrastructure; as the membership of the hosting set, processing capabilities and availability profiles of hosting machines, and client load change, our framework will adjust the number and placement of replicas to maintain the target quality of service.

Our management framework is novel in that it is completely distributed; each replica of a service is wrapped with a management agent that makes autonomous decisions about whether new replicas should be spawned and whether the current replica should be stopped or migrated to a better node. Agents rely on a set of loosely synchronized data to make their decisions, including the client load each replica is currently serving and load information about potential host machines. This autonomy and dependence only on loosely synchronized global state make our framework scalable and highly robust to the volatility inherent within federated systems. Despite autonomous actions based on weakly consistent data, we show that our management framework reaches a stable and appropriate configuration rapidly in response to system dynamics.

1.2 Contributions

Our contributions in this dissertation include:
• The study of a new paradigm for building moderately large scale distributed systems (i.e. 1000's of nodes). In particular, we adapt and create several probabilistic algorithms to construct an infrastructure for building and running federated services.

• The collection and study of several traces used to characterize four different environments for running federated services. Using these environments, we evaluate the performance of our infrastructure under different sets of constrains.

• The adaptation of existing gossiping algorithms to provide reliable communication and loosely consistent state replication on highly dynamic environments.

• The design of a scheme for searching and ranking documents distributively without building a centralized index. This scheme allows members to locate information as soon as it becomes online.

• Enabling nodes to autonomously replicate files in order to provide predictable data availability on heterogeneous environments. This work includes a novel use of erasure codes that reduces the overhead of repairing lost fragments.

• The design of a decentralized framework where autonomous agents monitor federated services to provide self-management and fault tolerance. Agents reduce the chances of operator induced faults by using application models to deploy, monitor and repair services.
Chapter 2

Related Work

This chapter describes some of the prior research related to this dissertation, focusing on other efforts that support the development of wide-area services.

2.1 Distributed systems and Internet computing

Since the 80’s researchers have studied distributed operating systems [68, 71, 78]. The introduction of local area networks (LANs) extended the role of traditional operating systems to deal with aggregating and virtualizing resources across machines. The new goal was not only to manage distributed resources, but also to provide new abstractions to simplify distributed programming. Systems like Sprite [71], Amoeba [68], and Locus [78] explored concepts like group communication, process migration, distributed file systems, etc.

Although we share similar goals in terms of functionality with these systems, our target environment is very different. Distributed operating systems were not designed to deal with issues like dynamic membership, large number of nodes, highly heterogeneous hardware, constant node and network failures, etc. These systems use solutions like centralized servers and distributed consistency algorithms that are not effective in federated environments (see Chapter 1).

Closer to our target environment, systems like Globus [30], Legion [43], Globe [92] and WebOS [98] address the problem of global scale computing. While Globe and Legion concentrate on the distributed programming paradigm, Globus and WebOS deal with the infrastructural problems.

All these frameworks use a structured approach toward managing information and
services. For example, Globus uses an LDAP [110] directory hierarchy to store information about the resources shared by a grid community. Similarly, Maarten van Steen et al. [93] builds his own hierarchical directory service to support mobile objects in Globe while optimizing for lookups and updates. WebOS adds more flexibility to the idea of a directory services by introducing Active Names [99]. An Active Name is a code snippet that is downloaded by a client the first time it tries to use a service (using DNS and a WWW server). To access the service, the client first runs the snippet which returns a service provider. Although Active Names have primarily been used for load balancing and transcoding on mobile environments, they could also be used to improve the client perceived availability of directory services.

In contrast, PlanetP uses the idea of unstructured communities to eliminate dependencies on individual nodes. PlanetP’s design reduces the need for hand customizing and configuring nodes according to their communal roles. On the other hand, PlanetP’s lack of structure limits its scalability as it increases the amount of information exchanged and maintained by each node. This decision is justified by the desire to support highly dynamic environments in which nodes may be available for small periods of time. In these environments a structured approach not only spends a significant amount time adding and removing nodes from the hierarchy, but it also requires complex algorithms to cope with instability [64].

2.2 Global scale infrastructural services

PlanetP uses a combination of gossiping and anti-entropy to achieve three main goals: providing probabilistically reliable multicast communication, managing communal membership and maintaining loosely synchronized global state. Object location is also provided by layering a global term-to-node index, which uses the loosely synchronized global state service.

Previous work has addressed some of these services although sometimes the goals and environments differ. For example, numerous research efforts have focused on building highly scalable distributed hash tables (DHT) for P2P communities [79, 85, 94, 113].
In general, DHTs use consistent hashing [58] to spread \( (key, value) \) pairs across the community and to provide retrieval mechanisms based on the \( key \). The idea of consistent hashing is that nodes are responsible for contiguous portions of the key space. This scheme is well suited for implementing distributed hash structures because node departures and arrivals are solved by just contacting a single existing node to release or acquire a portion of the key space (and its associated data). Although DHTs have been successfully used to build file system services [61, 69], we believe they are less suitable for the type of communities studied in this thesis. For example, the high cost of publishing thousands of keys and the lack of update propagation make it difficult to implement content-addressable publish/subscribe systems using only DHTs [51, 80]. PlanetP overcomes these difficulties using gossiping to propagate information and replicating a compact inverted index on every peer. Although this trade off limits scalability, it increases content publishing performance and fault-tolerance. Similarly, as membership becomes more dynamic the cost of adding and removing nodes (i.e. shifting keys) from a DHT starts to reduce the benefits of using a structured approach. Recent work [64] has shown that DHTs need special stabilization algorithms in order to preserve their invariants on highly dynamic communities.

In this thesis, we show that our combination of state replication and random gossiping provides bounded convergence times with no need for maintenance algorithms. Moreover gossiping has the advantage of distributing the cost of propagating new information across the community, thus helping nodes with limited resources.

The anti-entropy mechanisms used in PlanetP were originally introduced by Demers et al. [22]. The ideas behind these algorithms have been successfully applied to solve a variety of problems including membership management [47], information aggregation [100], multicast communication [9], database replication [22, 42, 76, 111], failure detection [101] and P2P DHTs [45]. As far as we know, nobody has looked at gossiping in scenarios where nodes join and leave constantly and in an uncontrolled manner. In this thesis we have quantified the use of gossiping techniques on P2P environments and adapted them for better bandwidth usage and propagation time stability (similar to the work done by Liben-Nowell et al. [64] for DHTs). Furthermore, Yu et al. [111] has shown
that the use of anti-entropy does not necessarily impose a weak consistency model. On their work, they study a consistency model and mechanisms that can drive anti-entropy to achieve a continuous consistency spectrum ranging from weak to strong consistency.

PlanetP’s information propagation and information sharing model was inspired by previous work on tuple spaces [34, 62] and publish/subscribe systems [13, 16, 27, 84]. Tuple spaces introduced the idea of space and time decoupling by allowing publishers to post tuples (i.e. data) without knowing the set of receiving nodes and, similarly, by letting receivers search for tuples at a later time. Publish/subscribe systems also added the concept of flow decoupling, meaning that nodes do not need to poll for updates. Instead, they are notified asynchronously when an event occurs. PlanetP is the first large scale decentralized infrastructure that has tried to leverage functionality from both of these bodies of work. Previous work in those areas had relied on assumptions like broadcast/multicast [86] or server based schemes [62] to communicate among members.

2.3 Autonomous Replication

Several solutions have been developed to increase data availability on wide area storage systems. A primary distinction of our work is that it considers availability for a wide range of communities that are distinct from those assumed by previous systems like CFS [21], Ivy [69], Farsite [1], and OceanStore [61]. In particular, we consider communities where the level of online dynamism is high and there may be considerable offline times for members operating in disconnected mode.

In contrast to our approach of randomly generating erasure coded fragments, OceanStore [61] proposes to replicate files using a fixed number of erasure coded fragments, which are repaired periodically but at a very low rate (on the order of once every few months [105]). In their scenario, nodes have relatively high availability and so replication, and fragment repair, are motivated mostly because of disk failures. Conversely, in the environments we study, the wide variance of node availabilities makes it difficult to adopt a similar approach and also requires a more pro-active replication algorithm.

Similarly, Ivy [69] uses a distributed hash table called DHash [21] to store file blocks
as well as a fixed number of replicas across a P2P community. To ensure availability, they assume that replicas are refreshed over time. Further, because the location of data in DHash depends on the current set of online members, it is difficult for them to adopt a less dynamic placement strategy such as ours. We believe that the constant refresh of data would be too expensive in highly dynamic communities.

We have found that the use of adaptive erasure codes and estimated node availability plays a key role in equalizing the overall file availability. Along these lines, Farsite [1, 11, 25, 26] uses information about node availability to increase minimum file availability. Although we have incorporated pieces of their algorithms into our replication scheme, there are several key differences. First, Farsite eliminates user-created file replicas before performing its internal replication, whereas we have to leave hoarded copies alone. Second, Farsite does not use erasure codes because it targets corporate LAN environments where average node availability is high compared to federated systems. Third, Farsite replicates all files equally while carefully placing them in order to maximize the minimum file availability, a metric that is quite meaningful for file systems. We, on the other hand, may replicate files at widely different levels to account for the different availability of peers hosting hoarded copies. In addition, we are more interested in the median/average file availability and the standard deviation than the minimum. This is because in our loosely coupled environments, there are always peers that are rarely online and so cannot replicate their content for high availability. The only option to increase these file’s availability is for more available peers to become interested in this content and start replicating it. Since we cannot control user interest, there may always be files with very low availability.

2.4 Content search and ranking

While current P2P systems such as Gnutella [37] and KaZaA [59] have been tremendously successful for music and video sharing communities, their search and information diffusion capabilities have been frustratingly limited. Our goal for PlanetP is to increase the power with which users can locate information in federated communities by providing content based search and ranking capabilities.
Several efforts parallel to PlanetP have also looked at better querying mechanisms [36, 49, 80]. Their focus, however, is on serving very large-scale communities. In order to be scalable these systems trade off performance and functionality by using iterative queries and distributed inverted indexes. None of this previous work supports content ranking.

More related to PlanetP’s information retrieval goals, Cori [14] and Gloss [40] address the problems of database selection and ranking fusion on distributed collections. Recent studies done by French et al. [32] show that both systems can scale to 900 nodes. Although Cori and Gloss use different indexing techniques, both maintain an index of all the nodes containing a term t and its number of occurrences at each node\(^1\). Using this data, they construct a centralized index that can direct queries toward the nodes storing the most relevant documents.

Because PlanetP focus larger and more dynamic communities, which do not have any centralized resources, we have chosen to keep even less information in the global index. Reducing the index size, allows us to minimize communication as well as storage. In spite of this minimal index, we find that our distributed search and rank algorithm is nearly as effective as the one used in a centralized Internet search engine like Altavista.

2.5 Federated Services Management

Previous research [4, 31, 74, 77, 91, 98, 100] have looked at building wide area infrastructures to simplify application development, software deployment and services management. Issues like group communication [74, 82, 91], service monitoring [31, 100], distributed data structures [94, 112], autonomic deployment [4, 91] and remote execution [31, 98] have been extensively studied. The focus of our work is to raise the abstraction level at which systems operators currently manage federated services. As we have noted on the introduction (Chapter 1), operators are the most frequent individual cause of service failures.

In our approach we use an application model, which could be derived from a Service

\(^1\)Gloss actually keeps slightly more global information than this
Level Agreement (SLA), to constantly monitor and improve service configuration. We use autonomous agents to constantly optimize the application model and deploy new configurations.

Traditionally, the use of SLAs and quality of service metrics (QoS) has been applied on cluster environments [6,17,90] where new configurations were evaluated and deployed by a single node. In our framework each agent operates autonomously, therefore it is possible for several agents to instantiate a new configuration simultaneously. Concurrent deployments may interfere with each other, leading to too many or too few replicas. To address this problem, we introduce the idea of probabilistic serialization, which resembles Ethernet’s back off algorithm for avoiding transmission collisions.

Previous work on global service management and deployment has used centralized solutions [31,98] or relied on group communication to coordinate deployment agents [91]. Closer toward our decentralization goals, work done on distributed agent-based systems [2,53], like Archon [52], has explored the use of agents that rely only partial communal information and cooperate to reach global goals. Although these frameworks have been evaluated on WAN environments, their underlying assumption is that agents can build a distributed database schema using information from an agent registry. Thanks to this schema agents can direct queries to other nodes and inquire about the state of the community. As discussed before the volatility of the environments studied on this work, makes this approach less attractive.

More recently, scientists have combined work on peer-to-peer and agent-based systems to build more robust and unstructured frameworks [67,89]. Still, nobody has yet address problems that require the kind of coordination needed to maintain federated services. For example, previous work has looked at load balancing of embarrassingly parallel jobs on P2P communities [67] and location of data on distributed tuple spaces [89].
Chapter 3

Infrastructure

3.1 Overview

In this chapter, we study the design of PlanetP’s communication layer. Our approach is comprised of two major components: (1) an infrastructural gossiping layer to support the replication of shared data structures across groups of peers, and (2) a replicated global directory that summarizes the information shared by the community. As presented on Chapter 1, the global directory is used to provide a multidimensional keyword index that can be used to locate objects like files, services, etc. In order to build this index we globally replicate two small data structures: a membership directory and an extremely compact keyword index. All members agree to continually gossip about changes to keep these shared data structures updated and loosely consistent.

In particular, we seek to answer the following questions:

- Can gossip based communication provide predictable consistency for shared data structures? That is, when a change occurs, how long does it take to reach all on-line peers and does it consistently reach all on-line peers?

- How does dynamicity and scale affect propagation time?

- What are the storage and bandwidth requirements needed to run PlanetP? We evaluate PlanetP in constrained environments, like P2P communities, to define the lower end of the spectrum.

We use simulation and measurements from our prototype implementation to answer these questions. In particular, we show that the globally replicated database that implements the multidimensional keyword index, only requires a modest amount of storage.
Changes to replicated data consistently reach all on-line members within several minutes. Further, synchronizing this database using our gossiping algorithm requires only a modest amount of bandwidth, even for extremely dynamic communities with very high rates of change.

3.2 Gossiping

PlanetP uses gossiping to replicate shared state across groups of nodes in a community. PlanetP’s gossiping algorithm is a novel combination of rumor mongering and anti-entropy as previously introduced by Demers et al. [22] together with a partial anti-entropy algorithm that we found improved performance significantly for highly dynamic environments. Briefly, Demers et al.’s algorithm works as follows when synchronizing a shared data structure that is replicated globally. Suppose $x$ learns of a change to the replicated data structure. Every $T_g$ seconds, $x$ would push this change (called a rumor) to a peer chosen randomly from its directory; the directory is a data structure that describes all peers in the community and is itself replicated everywhere using gossiping (see Section 3.3). If $y$ has not seen this rumor, it records the change and also starts to push the rumor just like $x$. $x$ stops pushing the rumor after it has contacted $n$ consecutive peers that have already heard the rumor. The intuition behind Demers’s algorithm is that at the beginning, rumors are known by few nodes and therefore they propagate rapidly. As rumors become popular, chances are that a node picked randomly will already know about it. While an early stop on rumor propagation saves bandwidth, it also leaves a residual set of peers that do not hear about a rumor before it dies out.

In order to eliminate this residue, every so often, each peer performs an anti-entropy operation instead of rumoring. For example, in our implementation of this algorithm, every tenth round of rumoring (or if there is currently no new information to be rumored), a peer $x$ would send an anti-entropy message instead of a rumor. The anti-entropy message asks the target $y$ to send a summary of its entire directory to $x$. When $x$ gets $y$’s summary, $x$ parses it to see whether $y$ has more updated information. If so, then $x$ asks $y$ for the needed information. This combination of push rumoring and pull anti-entropy helps to reliably spread new information everywhere.
Unfortunately, in a dynamic environment, the time required to spread new information can become highly variable. This is because rapid changes in the membership leads individual peers to have a less accurate view of the directory, elevating the problem of residual peers that do not receive rumors before they die out. Increasing the anti-entropy frequency is a possible solution to ensure eventual propagation of all the updates. Unfortunately, anti-entropy is much more expensive in terms of network bandwidth than rumor. While rumors are only as large as the update they carry, an anti-entropy communication session is proportional to the community size.

Thus, we instead extend each push operation with a partial pull that works as follows. When \( x \) sends a rumor to \( y \), \( y \) piggybacks the summaries of a small number \( m \) of the most recent rumors that \( y \) learned about but is no longer actively spreading onto its reply to \( x \); this allows \( x \) to pull any recent rumor that did not reach it. This partial pull requires only one extra message in the case that \( y \) knows something that \( x \) does not since the normal rumor operating process is really implemented as a query/request/reply sequence using unique rumor identifiers to save bandwidth when the target has already received the rumor. Furthermore, the amount of data piggybacked on \( y \)'s message is of constant size, on order of tens of bytes.

Observe that while the pushing of rumors has a termination condition, pulling does not. To address this, PlanetP dynamically adjusts its gossiping interval \( T_g \); if a peer is not actively pushing any rumors, it slowly raises its \( T_g \) (to some maximum value). When it receives a new rumor, it immediately resets its gossiping interval to the default. This dynamic adaptation leads to negligible bandwidth usage shortly after global consistency has been achieved.

Finally, note that although in this thesis, we assume that shared data structures are universally replicated and are gossiping with a single \( T_g \) for simplicity, this is not the general case. In fact, our implementation allows each data structure to be associated with only a subset of peers and gossiping at a distinct rate. This allows partial replication as well as rapid dissemination of time-sensitive information such as messages for group communication without increasing the overhead of maintaining more slowly changing data structures.
3.3 The Global Directory

In order to provide communal membership management and keyword based object lookup, PlanetP maintains a global directory. The global directory is a communal database that keeps track of all the members in a community. This database is replicated on every node using the gossiping layer described earlier.

Each record in the global directory represents a single peer and stores three types of information: first, a mandatory set of properties like IP address, user’s nickname, etc. Second, an optional set of numerical properties and third a set of terms associated with the objects shared by the peer (i.e. files, services, etc.).

In order to join a PlanetP community, new nodes need to contact an existing online member to get a copy of the current global directory. Using this copy, the new node adds itself to the directory and starts to propagate the updated version. Similarly, to advertise local objects a node just updates its corresponding entry on the local copy of the global directory and let the gossiping layer propagate it.

Thanks to the global directory, nodes can execute boolean queries to locate shared objects. A searching node first looks at all the terms associated with each entry to derive a group of nodes that contains these query terms. Then, it forwards the query to these nodes and asks them to return contact information for any object that is relevant to the query (i.e. URLs or RMI stubs). Effectively, the sets of terms included on the global directory provide a communal term-to-node index (referred as the global index). PlanetP uses this two-stage search process to perform exhaustive searches while limiting the size of the globally replicated index.

In order to minimize the amount of memory and bandwidth taken up by the global index, each node’s term set is implemented using a Bloom filter [10]. Briefly, a Bloom filter is an array of bits used to represent a set of strings; in our case, the set of terms associated with a node. The filter is computed by using \( k \) different hashing functions to compute \( k \) indices for each term and setting the bit at each index to 1. Given a Bloom filter, we can ask, if some term \( t \) is a member of the set by computing the \( k \) indices for \( t \) and checking whether those bits are 1. Bloom filters can give false positive answers but
never false negatives.

We use the combination of Bloom filters and a two stage search technique to implement the global multidimensional index, because it provides several advantages:

- Bloom filters are an efficient summary mechanism; for example, to support a false positive rate of less than 5% when using two hash functions, we would need 1.9 KB to summarize 1,000 terms and 34 KB for 50,000 terms. Furthermore, Bloom filters can be compressed and versioned to achieve a single bit per word average ratio when send over the network.

- Locally updating the global index with a new Bloom filter requires constant time, regardless of the number of changes introduced.

- Peers can independently trade-off accuracy for storage. For example, a peer \( x \) may choose to combine the filters of several peers to save space; the trade-off is that \( x \) must now contact this set of peers whenever a query hits on this combined filter. This ability for independently trading accuracy for storage is particularly useful for peers running on memory-constrained devices (e.g., hand-held devices).

- Peers can tell that documents relevant to a query might exist on nodes that are currently off-line. Thus, instead of missing these documents as in current systems, the searching peer could arrange to rendezvous with the off-line nodes when they reconnect to obtain the needed information.

In the following chapters, we will further extend the use of the global directory thanks to the optional per node properties. As shall be seen, we can use them to propagate node statistics like CPU load, node availability, available memory, etc.

3.4 Performance

Having described PlanetP’s data gossiping layer, we now turn to evaluating its performance. We study the cost (space and time) and the reliability of the gossiping layer when supporting the global directory. Our performance study is simulation-based but
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU gossiping time</td>
<td>5ms + (transfer-time × no. bytes)</td>
</tr>
<tr>
<td>Base gossiping interval</td>
<td>30sec</td>
</tr>
<tr>
<td>Max gossiping interval</td>
<td>60sec</td>
</tr>
<tr>
<td>Network BW</td>
<td>56Kb/s to 45Mb/s</td>
</tr>
<tr>
<td>Message header size</td>
<td>3 bytes</td>
</tr>
<tr>
<td>1000 terms BF</td>
<td>3000 bytes</td>
</tr>
<tr>
<td>20000 terms BF</td>
<td>16000 bytes</td>
</tr>
<tr>
<td>BF summary</td>
<td>6 bytes</td>
</tr>
<tr>
<td>Peer summary</td>
<td>48 bytes</td>
</tr>
</tbody>
</table>

Table 3.1: Constants used in our simulation of PlanetP’s gossiping algorithm.

most of the parameters were derived from a prototype implementation. Table 3.1 lists these parameters. We validated our simulator by comparing its results against numbers measured on a cluster of eight 800 MHz Pentium III PCs with 512MB of memory, running a Linux 2.2 kernel and the BlackDown JVM, version 1.3.0. Because of the JVM’s resource requirements, we were limited to 25 peers per machine, allowing us to validate our simulation for community sizes of up to 200 peers.

To drive the experiments, we study the replication of the global directory when used in the context of content search. In this environment, nodes parse and index every term on a document. Therefore, this is one of the most challenging scenarios as the number of terms per text file is very large when compared to other types of files. Events that change the directory and so require gossiping include the joining of a new member, the rejoin of a previously off-line member, and a change in a Bloom filter. We do not gossip the leaving (temporary or permanent) of a peer. Each peer discovers that another peer is off-line when an attempt to communicate with it fails. It marks the peer as off-line in its directory but does not gossip this information. When the peer \( x \) comes back on-line, its presence will eventually be gossiped to the entire community; each peer that has marked \( x \) as off-line in its directory changes \( x \)’s status back to on-line. If a peer has been marked as off-line continuously for \( T_{Dead} \) time, then all information about it is dropped from the directory under the assumption that the peer has left the community permanently.
3.4.1 Storage Cost

First, we assess the amount of memory required to store the global directory and then we focus on the networking costs of keeping it synchronized. In particular, we study the overhead introduced by the global index because it accounts for 98% of the global directory size in environments like the ones studied here, where mostly text files are shared.

**Global Index Size.** First, we study the space required by the global index to summarize the well known TREC [48] document collection (944,651 documents, 256,686,468 terms, 592,052 unique terms, 3,428.41 MB). This collection contains only text documents, so the ratio of unique terms to collection size is very high. For collections including multi-media documents, this ratio is likely to be much smaller. For example, a collection of 326,913 MP3 files requiring 1.4TB of storage collected from an existing P2P community only yielded 55,553 unique terms.

The size of the global index is determined by the summation of the number of unique terms per node. Therefore we need to address the effect of the community size, the document distribution and the number of documents per node. In the following experiments we assume a uniform document to node distribution. This distribution is a worst case scenario since any other document distribution (e.g. Weibull) would likely reduce the chances of having the same unique terms repeated across several nodes. For example, the best case scenario is when all the documents are stored on a single node.

Figure 3.1 shows that if a node already contains 0.4% of the TREC collection, it would have had to add approximately 3000 more documents, totaling 800,000 more terms, to have found an additional 1000 unique terms. The trend we found in Figure 3.1 is consistent with that found by a much larger study of word distribution [106]. These trends argue that sharing only unique terms across nodes is a scalable approach. Figure 3.1 shows that the rate at which new terms are introduced decreases with the number of documents per node.

Finally, in Figure 3.2 we count the number of unique words at each peer and compute the size of the global index if each Bloom filter was sized to summarize the per-node
unique terms with less than 5% probability of error. We also show what happens if each
document is replicated 3 times in the community as well as the case when we save space
by indexing only the 30% most frequent unique terms in each document (on Chapter 5
we show that this reduction still provides reasonable search results).

Observe that at 1000 peers, the global index is quite small: 16.1MB, which is just
0.5% of the collection. If each document were replicated 3 times, the storage requirement
would increase to 28.7MB, which is actually only 0.3% of the enlarged collection. At
5000 peers, the storage cost is somewhat higher, rising to 62.3MB if each document is
replicated 3 times. Observe, however, that if we sacrifice a little accuracy by indexing
only the 30% most frequent unique terms in each document, the storage requirement is
reduced again to 26.9MB, which is just 0.3% of the replicated collection.

Based on these results, we conclude that PlanetP should easily scale to several
thousand peers in terms of the required per peer storage for the replicated global index.

3.4.2 Propagating new information

We now assess the reliability and scalability of PlanetP’s gossiping algorithm. By relia-

city, we mean does each change propagate to all on-line peers? We begin by studying
Figure 3.2: Estimating the size of the global index when the TREC collection is uniformly distributed across a community of N peers. Each group of two bars shows, from left to right, the average number of unique words found on each peer and the size of the global index (in KB) if individual Bloom filters were big enough to summarize the per-node unique terms with at most 5% probability of error. Each bar is named after the community size, the replication factor (R1 or R3), and the percentage of per-document unique terms indexed.

The time required to gossip an update to the global directory throughout stable communities of various sizes. Measuring propagation time is important because it represents the window of time where peers’ directories are inconsistent, so that some peers may not be able to find new (or modified) documents.

In this experiment, we use a Bloom filter with 1000 words. Because PlanetP sends diffs of the Bloom filters to save bandwidth, this scenario simulates the addition of 1000 new terms to some peer’s local index. Note that, while 1000 new terms may seem small, it actually is quite large when comes from a node that already shares some documents as seen on Figure 3.1.

Figure 3.3(a) plots the time\(^1\) required for updates to reach every single peer on six different scenarios. These scenarios present different combinations of bandwidth capacity and gossiping schemes as described next:

**LAN** Peers are connected by 45 Mbps links. Peers use PlanetP’s gossiping algorithm.

**LAN-AE** Peers are connected by 45 Mbps links. Peers use only push anti-entropy:

---

\(^1\)These results are averaged across 10 independent runs.
each peer periodically push a summary of its data structure. The target requests all new information from this summary. This approach has been successfully used to synchronize smaller communities in Name Dropper [47], Bayou [23] and Deno [60].

**DSL-10,30,60** Peers are connected by 512 Kbps links. Peers use PlanetP’s gossiping algorithm. Gossiping interval is 10, 30, and 60 seconds respectively.

**MIX** Peers are connected by a mixture of link speeds. Using measurements of the Gnutella/Napster communities reported by Saroiu et al. [88], we create a mixture as follows: 9% have 56 kbps, 21% have 512 kbps, 50% have 5 Mbps, 16% have 10 Mbps, and 4% have 45 Mbps links.
Unless noted all the environments use a 30 second gossiping interval. Figure 3.3(b) shows the aggregate network volume used to propagate the new piece of information throughout the community. Figure 3.3(c) shows the average gossiping bandwidth used per peer during the experiment for DSL-10, DSL-30, and DSL-60.

Based on these graphs, we make several observations:

• **Propagation time is a logarithmic function of community size, implying that gossiping is very scalable to disseminate new information.** For example, propagation time for a community with 500 peers using DSL-30 is about 200 seconds, rising to only 250 for a community with 5000 peers. Although the scalability of gossiping is known to be logarithmic [22], it is important to verify that partial anti-entropy has not affected it.

• **Even though a change is gossiped throughout the entire community, the total number of bytes sent is very modest, again implying that gossiping is very scalable.** For example, propagation of a 1000 new terms throughout a community of 5000 peers requires an aggregated total of about 100 MB to be sent, leading to a per-peer average bandwidth requirement of less than 100 Bps when the gossiping interval is 30 seconds.

• **We can easily trade off propagation time against gossiping bandwidth by increasing or decreasing the gossiping interval.**

• **Our algorithm significantly outperforms ones that use only push anti-entropy for both propagation time and network volume.** Using rumoring enables PlanetP to reduce the amount of information exchanged between nodes while the mixture of pull (anti-entropy) and push (rumors) rounds reduces convergence time. On average, LAN-AE requires 2.3 times the network volume and twice the propagation time when compared to LAN.
Figure 3.4: *Time required for \( x - 1000 \) peers to simultaneously join the community of 1000 stable online peers, each wishing to share 20000 terms.*

### 3.4.3 Joining of new members

We now assess the expense of having groups of new members simultaneously join an established community. This represents the transient case of a rapidly growing community and is the worst case for PlanetP because each of these new members has to download the entire global index. Our simulator currently assumes that each client is single-threaded. Thus, a new member that is busy downloading the global index for a long time can cause significant variation in the propagation time of changes; this member cannot receive gossip messages while it is busy downloading.

In this experiment, we start a community of \( n \) peers and wait until their views of membership is consistent. Then, \( m \) new peers will attempt to join the community simultaneously. We measure the time required until all members have a consistent view of the community again as well as the required bandwidth during this time. For this experiment, each peer was set to share 20,000 terms with the rest of the community through their Bloom filters. Using the scenario presented on Section 3.4.1 we find that in order to index 20,000 unique terms, peers need to share 500MB of pure text documents.

Figure 3.4 plots the time to reach consistency vs. the number of joining peers for an initial community of 1000 nodes. These results show that if there is sufficient bandwidth (i.e. LAN), consistency is reached within approximately 600 seconds (10 minutes), even
when the community grows by 25%. In contrast to propagating a change, however, the joining process is a much more bandwidth intensive one; a joining member must retrieve 1000 Bloom filters representing a total of 20 million terms from the existing community. Also, having 250 members join at once means that 250 Bloom filters representing 5 million terms must be gossiped throughout the community. As a result, convergence times for communities interconnected only with DSL-speed links are approximately twice that of LAN-connected communities. Finally, convergence times for the MIX-connected communities become unacceptable, possibly requiring from 50 minutes to over two hours.

We draw two conclusions from these results. First, even in this worst-case scenario for PlanetP, which we do not expect to occur often, if peers have DSL or higher connectivity, then PlanetP does quite well. Second, we need to modify PlanetP if we are to accommodate users with modem-speed connections. While the artificial lengthening of gossiping convergence time can be easily fixed if peers are assumed to be multi-threaded, when a new peer first join, the time to download the entire directory would still likely take too long. Thus, we should either exclude peers with less than DSL connectivity or allow a new modem-connected peer to acquire the directory in pieces over a much longer period of time. We would also need to support some form of proxy search, where modem-connected peers can ask peers with better connectivity to help with searches.

We also decided to modify our gossiping algorithm to be bandwidth-aware, assuming that peers can learn of each other’s connectivity speed. The motivation for this is that a flat gossiping algorithm penalizes the community to spread information only as fast as the slow members can go. Thus, we modify the basic PlanetP gossiping algorithm for peers with faster connectivity to preferentially gossip with each other and peers with slower connectivity to preferentially gossip with each other. This idea is implemented as follows. Peers are divided into two classes, fast and slow. Fast includes peers with 512 Kb/s connectivity or better. Slow includes peers connected by modems. When rumorizing, a fast peer makes a binary decision to talk to a fast or slow peer. Probability of choosing a slow peer is 1%. Once the binary decision has been made, the peer chooses a particular peer randomly from the appropriate pool. When performing anti-entropy, a
Figure 3.5: CDF of gossiping convergence time in a community of 1000 when there are 100 Poisson arrivals (New arrivals share 1000 keys). LAN-NPA is our gossiping algorithm without the partial anti-entropy component.

fast peer always chooses another fast peer. When rumoring, a slow peer always chooses another slow peer, so that it cannot slow down the target peer, unless it is the source of the rumor; in this case, it chooses a fast peer as the initial target. Finally, when performing anti-entropy, a slow peer chooses any node with equal probability.

In the next section, we study the performance of the bandwidth aware algorithm when used in a dynamic environment where membership changes constantly.

3.4.4 Dynamic operation

Finally, we study the performance of PlanetP's gossiping when a community is operating in steady state, with members rejoining and leaving dynamically but without massive, simultaneous joins of new peers needing the entire global index. We expect this to be the common operational case for PlanetP. We begin by studying the potential for interference between different rumors as peers rejoin the community at different times. This experiment is as follows. We have a stable community of 1000 on-line peers; 100 peers join the community according to a Poisson process with an average inter-arrival rate of once every 90 seconds. Peers are connected at LAN speed. Each on-line peer has a Bloom filter with 1000 terms that off-line peers do not have. Each joining peer shares a Bloom filter with 1000 terms. Again, this represents the case where off-line
peers will have some new information to share, but they have to collect new information that may have accrued since they have been off-line. Figure 3.5 plots the cumulative percentage of events against the convergence time—the time required for an arrival event to be known by everyone in the on-line community—for PlanetP’s gossiping algorithm against what happens if the partial anti-entropy is not included. Observe that without the partial anti-entropy, overlapping rumors can interfere with each other, causing much larger variation in the convergence times.

To complete our exposition, we study a dynamic community with the following behavior. The community is comprised of 1000 members. 40% of the members are online all the time. 60% of the members are online for an average of 60 minutes and then offline again for an average of 140 minutes. Both online and offline times are generated using a Poisson process. 20% of the time, when a peer rejoins the on-line community, it sends a Bloom filter diff containing 1000 new terms. These parameters were again based roughly on measurements reported by Saroiu et al. [88] (except for the number of new terms being shared occasionally) and are meant to be representative of real communities. We note again that 1000 new unique terms typically represents the sharing of a significant set of new documents. (We have also studied a more dynamic community, where 50% of the time, a peer coming back on-line shares 100 new words. The results are similar to those present below.)

Figure 3.6(a) plots the cumulative percentage of events against the convergence time. We observe that with sufficient bandwidth, convergence time is very tight around 400 seconds. For the MIX community we separate the CDF in two classes: the time it takes for fast nodes to propagate events to other fast nodes (MIX-F) and the time it takes for slow nodes to reach the whole community (MIX-S). Intuitively, what we are trying to assess is whether the presence of slow nodes affects the performance of the well connected part of the community when using the bandwidth aware algorithm (BA). The graph shows that the BA algorithm allows fast nodes (MIF-F) to propagate events as in the LAN case.

Furthermore, in Figure 3.6(b) we compare the convergence time for the non bandwidth aware algorithm (NBA) against the BA version. The figure shows that BA does
not harms the propagation speed for slow nodes (MIX-S vs. MIX-S NBA), but it significantly help the fast nodes (MIX-F vs. MIX-F NBA) to propagate at their maximum speed.

Finally, Figure 3.7 plots the aggregate bandwidth against time. This graph shows that the normal operation of a community requires very little bandwidth, ranging from between 10 KB/s to 100 KB/s across the entire community.

3.5 Summary

PlanetP uses gossiping to robustly disseminate new information and to maintain a loosely replicated database across all the nodes.

We have shown that changes to replicated data consistently reach all on-line members within several minutes even when the communal membership changes at a fast speed. Further, synchronizing a global database, like the global directory, using our gossiping algorithm requires only a modest amount of bandwidth. We have used a challenging environment, where a heterogeneous P2P community shares only text documents and is constantly adding new files to predict performance in a worst case scenario.
Figure 3.7: Aggregated bandwidth usage during the normal operation of a dynamic community with 1000 members.
Chapter 4
Data availability

4.1 Overview

In this chapter we consider the problem of content availability for file sharing communities. Our goal is to autonomously replicate information, such that highly dynamic communities where individuals may be disconnected as often as being online, can efficiently control content availability. Our approach explores the idea of placing replicas in a way that, despite constant changes to the online membership, files are highly available without requiring the continual movement of data. At first, we assume that nodes eventually rejoin the online community when they go offline to reduce the amount of replication, although we later relax this assumption. We propose and evaluate a distributed replication algorithm where all replication decisions are made autonomously by individual members using only a small amount of loosely synchronized global state shared through PlanetP’s global directory. To maximize the utility of excess storage, we assume that files are replicated in their entirety only when a member hoards that file for disconnected operation. Otherwise, files are replicated using an erasure code [104]. While the use of an erasure code is not novel, we show how to avoid the need for tracking the placement of specific fragments. In particular, we show how to increase the availability of a file by adding new random fragments rather than regenerating individual fragments that have been lost.

We deliberately study a very weakly structured system because tight coordination is likely difficult and costly in large distributed and dynamic communities [64]. Fundamentally, our approach only depends on nodes managing their individual excess storage in a fair manner and having approximate data about replica-to-node mapping and average node availability. The need for information on replica-to-node mapping is obvious.
With respect to node availability, without this information, a replication algorithm cannot differentiate between nodes with very different availabilities and thus it may under- or over-replicate files. Beyond this loosely synchronized global state, all decisions are local and autonomous. Of course, one can increase the level of coordination between nodes to increase the efficiency of the system in utilizing storage and bandwidth. In essence, we seek to conservatively estimate the amount of storage required to provide a practical availability of 99.9%, which, as already mentioned, is comparable to today’s web services.

Our studies show that it is possible to increase availability of shared data to practical levels, e.g., 99.9%, using a decentralized algorithm where members operate autonomously with little dependence on the behaviors of their peers. We study the performance of our algorithm for three distinct environments, modeled using data published from earlier studies of a corporate environment [11], the Napster and Gnutella file-sharing communities [88], and a file sharing community local to students of our University.

4.2 Autonomous Replication

In our replication approach, each member of a community hoards some subset of the shared files entirely on their local storage, called the member’s hoard set, and pushes replicas of its hoard set to nodes with excess storage using an erasure code. We propose such a structure to support disconnected access to shared data. In loosely organized applications such as current file sharing, hoarding is uncoordinated and entirely driven by members’ need for disconnected access. In more tightly organized applications, such as a file system, the application can coordinate the division of the shared data set among individuals’ hoard sets, in essence dividing the responsibility for ensuring the availability of the shared data.

To simplify our description, we introduce a small amount of terminology. We call a member that is trying to replicate an erasure-coded fragment of a file the replicator and the node that the replicator is asking to store the fragment the target. We call the excess storage space contributed by each member for replication its replication store. (Note
that we do not include replication via hoarding as part of the replication store.) We assume that each file is identified by a unique ID. Finally, when we say “the availability of a fragment,” we are referring to the availability of the file that the fragment is a piece of.

Given this terminology, the overall algorithm is as follows:

- Each member advertises the unique IDs of the files in its hoard set and the fragments in its replication store in the global index. Each member also advertises its average availability in the global directory.

- Each member periodically estimates the availability of its hoarded files and the fragments in its replication store.

- Periodically, say every $T_r$ time units, each member randomly selects a file from its hoard set that is not yet at target availability and attempts to increase its availability; the member does this by generating a random erasure coded fragment of the file and pushes it to a randomly chosen target.

- The target accepts and saves the incoming fragment if there is sufficient free space in its replication store. If there is insufficient space, it either rejects the replication request or ejects enough fragments to accept the new fragment. Victims are chosen using a weighted random selection process, where more highly available fragments are more likely to be chosen.

Our goal in designing this algorithm is to increase the availability of all shared files toward a common target availability while allowing nodes to act completely autonomously using only a small amount of loosely synchronized global data. Given a replication store that is very large compared to the set of documents being shared, we know that this approach will work [83]. The interesting questions become what is the necessary ratio of the replication store to the size of the document set and what happens when the replication store is not sufficiently large for the community to achieve the target availability for all files. We explore these questions in Section 4.3. In the remainder of this section, we will discuss our use of erasure coding, how to estimate file
availability, our replacement policy, and the resiliency of our approach to misbehaving nodes.

### 4.2.1 Randomized Fragmentation and Replication

We use the Reed Solomon (RS) erasure coding in a novel way to support autonomous member actions. The basic idea of any erasure code is to divide a file into $m$ fragments and recode them into $n$ fragments, where $m < n$, in such a way that the file can be reassembled from any $m$ fragments with an aggregated size equal to the original file size [81].

To date, given $(n, m)$, most uses of erasure codes generate all $n$ fragments and, over time, detect and regenerate specific lost fragments. This approach has three significant disadvantages for highly dynamic environments: (i) as the average per-member availability changes over time, files’ availability will change given a fixed $n$; to maintain a target availability, it may thus be necessary to change $n$, necessitating the re-fragmenting and replication of some (perhaps all) files; (ii) an accurate fragment-to-node mapping is required for the regeneration of fragments lost due either to nodes leaving the community permanently or ejection from the replication store; and (iii) it must be possible to differentiate accurately between nodes temporarily going offline and leaving the community permanently to avoid introducing duplicate fragments, which reduces the effectiveness of erasure coding.

To overcome these limitations, we choose $n >> m$ but do not generate all $n$ fragments. When a member decides to increase the availability of a file, it simply generates an additional random fragment from the set of $n$ possible fragments. If $n$ is sufficiently large, the chances of having duplicate fragments should be small, thus maximizing the usefulness of every fragment generated in spite of not having any node coordination. In this manner, it is easy to dynamically adjust the number of fragments generated for each file to reflect changes in the community.

RS is particularly suited to our proposed use because the cost of generating each fragment is independent of $n$. The fundamental idea behind RS is that a polynomial of degree $m − 1$ in a Galois field $GF(2^n)$ is uniquely defined by any $m$ points in the
field. In order to create an erasure code for the blocks $D_1, ..., D_m$ of a file, we need a polynomial $p$ such that $p(t_1) = D_1, ..., p(t_m) = D_m$. Once we have this polynomial, it is easy to create up to $2^w - m$ additional points $p(t_i) = D_i$, $i > m$ such that the polynomial can be reconstructed from any combination of $m$ items from the set $\{(t_1, D_1), ..., (t_m, D_m), ..., (t_i, D_i), ...\}$. Observe that, given the polynomial, the generation of any one fragment is independent of $n$ as desired. According to Rizzo [81], files can be encoded/decoded on a Pentium 133Mhz at 11MB/s. Moreover using $w$’s up to 16 is quite feasible, which translates into a 0.006 probability of having collisions for the environments studied in Section 4.3.

4.2.2 Estimating the Availability of Files

To provide a global mapping of file placement, whenever a member $m$ hoards a file $f$, it constructs a term that is a concatenation of the word $File$ and a hash of $f$’s content $Hash(f)$, and inserts the mapping $File \_ Hash(f) \rightarrow m$ into the global index. Likewise, if $m$ accepts a fragment of file $f$ for storage, it inserts the mapping $Frag \_ Hash(f) \rightarrow m$ into the global index. These mappings are of course removed if $m$ stops hoarding $f$ or evicts the fragment of $f$ that it previously accepted. We further assume that nodes advertise their average online and offline times in the global directory.

Given the above information in the global index, we can identify the set of nodes hoarding any file $f$, $H(f)$, and those that contain a fragment of $f$, $F(f)$. Then, assuming that nodes’ behaviors are not correlated [7, 11], the availability of $f$, $A(f)$, can be estimated as 1 minus the probability that all nodes in $H(f)$ are simultaneously offline and at least $n - m + 1$ of the nodes in $F(f)$ are also offline; $m$ is the number of fragments required to reconstruct $f$ and $n$ is redefined as $n = |F(f)|$. In general, since every node in the system may have a different probability for being offline, say $P_i$ for node $i$, it would be too expensive to compute the exact file availability. Thus, we instead use the following approximation that uses the average probability of being offline ($P_{avg}$) of nodes in $F(f)$:

$$A(f) = 1 - \prod_{i \in H(f)} P_i \sum_{j=n-m+1}^{n} \binom{n}{j} P_{avg}^j (1 - P_{avg})^{n-j}$$ (4.1)

where
\[ P_i = \frac{\text{avg. offline time}}{(\text{avg. online time} + \text{avg. offline time})} \]

\[ P_{avg} = \frac{1}{n} \sum_{i \in F(f)} P_i \]

Note that equation 4.1 assumes that \( H(f) \) and \( F(f) \) do not intersect and that all \( n \) fragments reside on different nodes. We ensure this by allowing a node to either hoard an entire file or store only a single fragment of that file.

In addition, we are assuming a closed community, where members may go offline but do not permanently leave the community. Currently, we assume that members will be dropped from the directory if they have been offline for some threshold amount of time. Thus, when members permanently leave the community, the predicted availabilities of files may become stale. Since we periodically refresh the predicted availability of files, this should not become a problem.

Finally, note that equation 4.1 does not account for the possibility of duplicate fragments; as already argued, however, we can make the chance of having duplicate fragments quite small and so the impact should be negligible.

### 4.2.3 Replacement

When a target node receives a replication request, if its replication store is full, it has to decide whether to accept the incoming fragment, and if it does, select other fragments to evict from its replication store. Because we are trying to equalize the availability of all files across the system, we would like to eject fragments with the highest availability. However, if we use a deterministic algorithm then multiple nodes running the same algorithm autonomously may simultaneously victimize fragments of the same file, leading to drastic changes in the file’s availability. Thus, we instead use a weighted random selection process, where fragments with high availability have higher chances of being selected for eviction.

Our replacement policy is as follows. We first compute the average number of nines in the availability of all fragments currently stored at the target. Then, if the incoming fragment’s number of nines is above 10% of this average, we simply reject it outright.
Otherwise, we use lottery scheduling [103] to effect the weighted random selection of victim fragments. In particular, we create a set of tickets and divide them into two subsets with the ratio 80:20. Each fragment is assigned an equal share of the smaller subset. In addition, fragments with availability above 10% of the average are given a portion of the larger subset. The portion given to each fragment is proportional to the ratio between its number of nines and the sum of the number of nines of all such fragments. The notion of different currencies makes this division of tickets straightforward. For example: if a target node has three fragments with availabilities 0.99, 0.9, 0.5 or 2, 1, .3 “nines” respectively\footnote{No. \textit{nines} = \(-\log_{10}(1 - \text{availability})\)} then the average availability in nines plus 10% is 0.76. Now if we have 100 lottery tickets the first fragment will get 67 + 6.6 tickets from the first and second pool respectively, the second fragment will get 13 + 6.6 tickets and the third fragment will get 0 + 6.6 tickets. Overall, the probability of each fragment being evicted will be 0.73, 0.19 and 0.06 respectively.

The intuitions behind our replacement policy are as follows. First, we reject the incoming fragment if it will simply become a target for eviction the next time a replication request is received by the target node. Without this condition, we will simply be shifting fragments around without much effect. Our threshold for this outright rejection may seem rather low; at some cost of bandwidth, if we were less aggressive at rejecting fragments, perhaps over time, the system can reach a better configuration. However, our experimentation shows that while this threshold affects bandwidth usage, it does not significantly affect the overall replication. Since we are targeting environments where bandwidth may be a precious commodity (see Section 4.3), we decided that an aggressive threshold was appropriate.

Next, we penalize over-replicated files heavily for the number of nines in their availability, making it highly probable that a fragment of an over-replicated file will be evicted. We use the number of nines rather than the availability itself because it linearizes the differences between values, i.e. the difference between 0.9 and 0.99 is the same as that between 0.99 and 0.999.
4.2.4 Optimizations at Replicators

While the critical decisions rest at the targets in our algorithm, replicators can implement several optimizations to increase the convergence rate. First, a replicator might increase availability convergence speed by favoring files that have low estimated availability over files with high estimated availability. Because of hoarding and potentially stale data, it is again necessary to use a weighted random selection process instead of a deterministic process. We use lottery scheduling in a manner similar to replacement, except in the reverse sense, of course, favoring files with low availability.

Second, a replicator can try to find nodes with free space in their replication store rather than causing an eviction at a node whose replication store is full. To implement this optimization, a replicator first selects a target randomly. Then, it inquires whether the target has sufficient free space in its replication store. If not, then the replicator selects another target, again randomly. The replicator repeats this process for five times (it could arbitrarily be any number of times), and then gives up and simply chooses randomly from these five previously chosen targets.

Finally, a member can choose to replicate only a subset of its hoard set to increase the availability of this subset at the cost of reduced availability for the remaining files. The power of our approach is that, if some member is interested in a file being highly available, it can act as a champion for that file by hoarding it. Ultimately, it is up to the application that places the hoarded files to decide when extra copies are needed.

4.3 Evaluating the Achievable Availability

In this section we present a simulation-based study of our replication scheme’s impact on file availability. We begin by describing our simulator and three distinct communities that we will study. Then, we present simulation results and discuss their implications.

4.3.1 Experimental Environment

We have built an event driven simulator for this study. To achieve reasonable simulation efficiency, we made several simplifying assumptions. First, we assume that all members
of a community attempt to replicate files from their hoard sets at synchronous intervals. Second, we do not simulate the detail timing of message transfers and the full PlanetP gossiping protocol that is used to replicate the global directory and the global index. In order to account for potential data staleness due to the latency of gossiping, we reestimate file availability and reassign tickets for the various lotteries only once every 10 minutes\(^2\). This batching of computations actually serves two purposes. First, it forces nodes to work on outdated and inaccurate information (far more inaccurate than the 3 minutes latency provided by PlanetP on the communities studied on Chapter 3). Second, it simulates a practical real implementation since estimating files’ availability and giving tickets to every file and fragment is not a computationally trivial process.

Finally, we always fragment files using a Reed Solomon code with a fixed \(m\) set to 10. This means that all fragmented files can be reassembled with 10 fragments, regardless of its size. Although in practice the algorithm could vary \(m\) to achieve the best trade-off between fragment size, download latency, and space utilization, we fixed it to simplify the analysis.

### 4.3.2 Simulated Communities

We define three distinct P2P communities to assess the impact of replication under different operating environments representing different styles of communities. The first community is representative of a very loosely coupled community sharing mostly multimedia files; the second resembles a corporate department; and the third models a distributed development group. The environments vary in the following ways:

- The per node mean uptime and downtime; we model node arrival to and exit from the online community as exponential arrival processes based on the given means
- The number of files per node
- The number of hoarded replicas per file.

\(^2\)This simulated time unit is only relevant when compared with the gossiping interval, which is assumed to be 30 seconds. This 30 seconds interval can be thought of as the unit time. For example, if we map this unit to 60 seconds, then all time periods reported in our results would simply be multiplied by two.
<table>
<thead>
<tr>
<th>No. Members</th>
<th>File Sharing (FS)</th>
<th>Corporate (CO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1,000</td>
<td></td>
</tr>
<tr>
<td>No. Files</td>
<td>25,000</td>
<td>50,000</td>
</tr>
<tr>
<td>Files per Member Dist.</td>
<td>Weibull(sh:1.93,mean:25)</td>
<td>Weibull(sh:6.33,mean:50)</td>
</tr>
<tr>
<td>File Size Dist.</td>
<td>Sigmoid(mean:4.3 MB)</td>
<td>Lognormal(μ:12.2 σ:3.43)</td>
</tr>
<tr>
<td>Hoarded Replica Dist.</td>
<td>mod. Pareto(sh:0.55 se:3.25)</td>
<td>None</td>
</tr>
<tr>
<td>Node Availability</td>
<td>24.9% (average)</td>
<td>80.7% (average)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workgroup (WG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Members</td>
</tr>
<tr>
<td>Files per Member Dist.</td>
</tr>
<tr>
<td>File Size Dist.</td>
</tr>
<tr>
<td>Hoarded Replica Dist.</td>
</tr>
<tr>
<td>Node Availability</td>
</tr>
</tbody>
</table>

Table 4.1: Parameters used to simulate each environment.

- The amount of excess space on each node
- The file sizes. Table 4.1 summarizes the parameters of each of the communities.

For all three communities, we vary the amount of excess space provided by each member across several different simulations. In all cases we refer to excess space as a proportion of the number of bytes in members’ hoard sets. In all scenarios, the number of files and nodes has been scaled to allow us to run experiments within reasonable times. As shall be seen, since nodes provide excess space based on what they share, the total number of files and nodes will not affect the algorithm. In all experiments, we assume a target file availability of three nines (99.9%).

**File-sharing (FS).** The first community that we study is modeled using two sources of data:

- Saroiu et al.’s reported statistics collected from the Gnutella and Napster communities [88]
- Data collected from a local DirectConnect [24] community (DC) comprised of University students

In particular, for this community, we simulate 1,000 nodes sharing 25,000 files. The number of files per node is modeled using a Weibull distribution approximating Saroiu et al.’s reported actual distribution. File sizes are modeled using a Weibull distribution
approximating our measurements of the local student community, which is shown in Figure 4.1(a). We cross these data sets because Saroiu et al. do not report the latter. Since the communities serve essentially the same purpose, we believe that data from one is likely to be representative of the other (we verified this by checking characteristics that were measured in both studies). The number of hoarded replicas per file was also taken from the local community and is shown on Figure 4.1(b).

Finally, we use Saroiu et al.’s reported Gnutella uptimes and availabilities to drive members’ arrival to and departure from the online community. We made two assumptions because of insufficient data:

- We correlate availability with average length of online times; that is, the most highly available members also have the longest average online times. This correlation is intuitive because it avoids the pathological case of a highly available member constantly and rapidly flipping between being online and offline

- We correlate the file distribution with availability, where the most highly available members have the most files. This correlation is motivated by measurements on our local student community, and suggests that “server-like” nodes typically hold the largest portion of the shared data.
Figure 4.2: CDF of the number of files per user observed on DC and a fitted 2 parameter Weibull distribution (shape:0.69, scale:1167, mean:1598 files per user).

**Corporate (CO).** The second community represents what we might expect to see in a corporate or university environment. In addition to being a significantly different environment than FS, studying this environment also allow us to compare our results with Farsite [11]. Farsite has somewhat similar goals to ours but takes a significant, almost inverse approach (see section 2.3 for more details). Thus, this community is modeled using data provided by Bolosky et al. [11]. All the parameters shown in Table 4.1 were taken directly from Bolosky et al.’s work with the exception of the absolute number of nodes and total number of files, which were scaled down to limit simulation time.

**Workgroup (WG).** Finally, the third configuration tries to capture a geographically distributed work group environment, such as a group of developers collaborating on an open-source project. The idea is to evaluate the impact of not having any member with server-like behaviors, i.e., members with large hoard sets that are (relatively) highly available. In particular, we simulate 100 nodes sharing 10,000 files of 290KB each—this is the average size of non-media files found in [29]. The number of files per user is distributed either uniformly or according to a distribution approximating our measurements of the local P2P network, shown in Figure 4.2. Nodes will follow a work schedule with a mean of 1 hour online followed by 2 hours offline. Again, arrival is exponentially
distributed.

4.3.3 Other Replication Algorithms

To evaluate the impact of the amount of information assumed to be available to drive the replication algorithm, we will compare our algorithm, called REP, against two alternatives: the first one, called BASE, simulates nodes pushing and accepting replicas in the complete absence of information on file availability; the second, called OMNI, assumes centralized knowledge, where replication is driven by a central agent that tries to maximize the minimum file availability. Thus, a comparison with BASE quantifies the effect of having approximate knowledge of current file availability. Comparing against OMNI quantifies the effect of autonomous actions using loosely synchronized data as opposed to having perfect, centralized information.

In BASE, nodes will use FIFO to manage their excess storage and replicators will only select files when its estimated availability is below the target availability. Note that this is really just an optimization to limit simulation time as the results would be essentially the same if BASE didn’t implement this optimization.

OMNI is implemented as a hill-climbing algorithm that tries to maximize the availability of the least available file on every step. OMNI assumes centralized/perfect knowledge of node behaviors, file and fragment distributions, and the ability to replicate a fragment on any member at any time. This algorithm was motivated by Bolosky et al.’s work [11], who shows that in an environment like CO it produces allocations close to the optimal.

4.3.4 Availability Improvements

Availability of Nodes vs. Excess Storage Needed. We begin our evaluation considering two factors affecting file availability: the size of the aggregate replication store, and members’ average availability. In particular, we use equation 4.1 as a simple analytical model.

Figure 4.3(a) shows the file availability plotted as a function of available excess space for mean node availability equal to that of the three communities. As expected
Figure 4.3: *Achieved file availability in number of nines plotted against excess storage in terms of x times the size of the entire collection of hoarded files for (a) 0 replication from hoarding and (b) 25% of the files have five hoarded replicas.*

from [11, 25], only a small amount of excess space is required to achieve very high file availability in the CO community: 3 nines availability can be achieved with 1.75X excess capacity while almost 5 nines can be achieved with 2X excess capacity. Achieving high availability is much more difficult for FS-type communities because many nodes only go online briefly to locate and download information: around 8X of excess capacity is required for 3 nines availability. As shall be seen later, however, this model is pessimistic because using average node availability looses the importance of server-like nodes that are online most of the time. Our simulation results will show that only around 1.5X and 6X of excess capacity are needed to achieve 3 nines availability on CO and FS respectively.

Finally, we observe from figure 4.3(b) that file hoarding has little effect on file availability except in the case where node availability on average is very high. Although this is again pessimistic for communities with server-like nodes, it validates our decision to use erasure coding for replication (and is consistent with previous results obtained by Weatherspoon et al. [104], showing that erasure coded replication can reduce space usage by 2 orders of magnitude).

**Overall Availability.** We now consider simulation results for the three communities discussed earlier. Figure 4.4 shows the CDFs of file availability for all three communities; Table 4.4(d) pulls out some of the more interesting points in these CDFs. In these
Figure 4.4: CDFs of file availability for (a) the CO community with 1X, 1.5X and 2X, (b) the FS community with 1X, 3X, and 6X, and (c) the WG community with 9X excess storage capacities. Table (d) extracts the minimum and average availability and the 1% and 5% points, corresponding to the minimum availability of 99% and 95% of the most available files respectively.

...figures, and for the remainder of the paper, all presented file availability are computed using equation 4.1, using the final placement of files and fragments, with \( P_{avg} \) computed using the expected availability of each node. An alternative approach would have been to present the actual measured availability; that is, divide the observed amount of time that a file was accessible by the total simulated time. However, this estimation is often optimistic because it is difficult to simulate a sufficiently long period of time to observe...
significant downtime for highly available members.

As expected from our analytical study, the CO community has little trouble in achieving high file availability because the average availability of members is high. At 2X excess storage capacity, over 99% of the files have higher than 3 nines availability, with the minimum file availability of 2.73 nines (or 99.8%).

For the FS community, we see that using around 6X excess storage capacity, we achieve 3 nines availability for 99% of the files, with the minimum availability being 2.9 nines. This is significantly less than the 8X predicted by our model. Because of the presence of server-like nodes that are highly available, files that are hoarded by these nodes achieve high availability from the beginning without using extra storage. Further, any fragment stored at these nodes increases the availability of the corresponding file significantly and reduces the total number of fragments needed.

Observe that for both CO and FS, there are subsets of files that achieve much greater availability than the average. Interestingly, this is not related to the fairness properties of our algorithm. Rather, it is because these files are naturally replicated on very highly available nodes through hoarding. As you will see later on Figure 4.6(b), when we discuss REP’s space allocation, most of these files are not fragmented at all, or have only a few fragments, which are useless since files need at least 10 fragments to increase availability. This constitutes a key difference between our assumed environment, and thus the approach to replication, and a file system such as Farsite. In the latter environment, the “excess” hoarded copies would be collapsed into a single copy and thus increasing the fragment space; however, our view of hoarded copies as user-controlled copies prevents us from taking that approach.

For the WG environment, we observe an interesting effect. When files and excess storage are uniformly distributed among nodes, 9X excess capacity is enough to achieve 3 nines availability, as predicted. If files and excess capacity are non-uniformly distributed, however, then average availability drops sharply. This degradation, also observable on OMNI, arises from the small number of nodes in the community and our assumed per-node coupling between the size of the hoard-set and the replication store. In this case, nodes with the most files to replicate also have the most excess storage, which is not
useful to them. Further, since there are no high-availability nodes that are up most of the time, it is not easy for replicators to find free space on the subset of nodes with large replication stores. This points to the importance of even a few server-like nodes that provide both excess capacity as well as high availability in communities similar to FS.

**Comparing Against Having Centralized Knowledge.** Figure 4.4 also shows that having centralized information allows OMNI to: (i) achieve the target availability with less excess space, and (ii) increase the minimum availability when the target availability cannot be achieved for all files. While the former is not insignificant—for example, for CO, OMNI was able to achieve 4 nines availability with only 1.5X excess storage and 2.43 nines availability with only 3X excess storage for FS—we believe the latter is a more important difference. With the continual rapid growth in capacity and drop in price of storage, having sufficient excess space on order of 2-9X does not seem outrageous. However, increasing the minimum file availability may be critical for applications like file systems. It is interesting to note, however, that occasionally, REP can outperform the heuristic-based OMNI; for example, for WG 9X where documents are distributed according to a Weibull distribution, REP achieves a minimum availability of 1.5 whereas OMNI can only achieve 1.35.

One main reason why we cannot approach OMNI more closely for minimum availability in most instances, is that members with low availability cannot maintain high availability for files in their hoard sets when there is insufficient excess storage. Over time, nodes that are more available win the competition for excess storage because they have the advantage of being able to push their hoarded files more often. This is a direct consequence of our desire for autonomous node actions. One potential approach to alleviate this problem is for more highly available nodes to also hoard these files so that they can “lend a hand” in pushing replicas.

**Effects of Availability-Based Replacement.** To assess the importance of being able to estimate file availability during replacement, Figure 4.5 shows the CDFs of the expected file availability for BASE and REP on the FS community with 3X excess storage. Observe that the main effect of the receiver implementing a smart replacement
Figure 4.5: *Comparison of BASE and REP’s CDFs of file availability for the FS community 3X.*

Figure 4.6: *The number of fragments per file with the files ordered according to the average availability of the nodes that are hoarding them for (a) BASE, and (b) REP when running on FS 3X.*

algorithm is to increase fairness. Under BASE, nearly 16% of the files have availability less than a single nine, compared to only 1% under REP. Similarly, REP achieves a better average availability of 1.7 nines compared to 0.9 nines in BASE.

Figure 4.6(a,b) shows the reason behind these differences. BASE’s FIFO replacement strategy favors nodes that are frequently online. These nodes achieve faster push rates and so they obtain an unfair fraction of the replication store. Their files end up having many more fragments than those on less available nodes, even though the community needs the latter’s content to be more highly replicated. This effect is magnified when a file is hoarded by several nodes with high availability, because it gains an even greater
aggregated push rate. Under REP’s replacement policy, the replication store is divided more evenly, with some favoring of files from less available nodes.

In brief the effect of BASE’s uncontrolled replacement policy leaves the control of the excess space to the most available nodes, which are the ones that need it the least. **Bandwidth Usage.** Finally, we study the amount of bandwidth used after a community has reached stability. To put bandwidth in perspective, we will use the average number of bytes transferred per hour per node over the average file size. In essence, this metric captures the rate of replication that continues because, lacking centralized knowledge, the system is not able to discern that it should cease trying to increase availability—there is just not sufficient excess storage.

For CO, when the excess space is only 1X, REP continues to replicate on average 10 files per hour per node. If we increase the amount of excess storage to 2X the average amount of files replicated drops to zero, recall from figure 4.4(a) that 2X is enough to reach the target availability of 3 nines. Similarly, in FS the number of files replicated per hour per node goes from 241 to 0 as we increase the excess space from 1X to 3X (figure 4.4(b)). When comparing these results to the ones obtained by the BASE algorithm we find that for FS with 3X excess storage (figure 4.5), BASE replicates 816 files per hour against zero in REP.

This points out an important advantage of being able to dynamically estimate file availability: when the available excess storage is not too much less than what is required to achieve the target availability, our approach is relatively stable. This implies that while it is important to set the target availability to what the community can support—for example, it is not reasonable to target three nines availability when there is only 1X excess storage in FS as shown by the large number of files being pushed—it doesn’t have to be highly accurate.
4.4 Summary

In this chapter, we have addressed the question of increasing the availability of shared data for a range of distributed communities. We have attempted to quantify a conservative estimate of the amount of excess storage required to achieve a practical availability of 99.9% by studying a decentralized algorithm that only depends on a modest amount of loosely synchronized global state. Indeed, our results show that it is exceedingly difficult to achieve this level of availability if individuals do not at least have approximate knowledge of nodes’ availability and files’ current availability, which exactly comprise the global state that we assume are maintained by our infrastructure.

Our results are extremely encouraging because they demonstrate that practical availability levels are achievable in a completely decentralized P2P system with low individual availability. For example, even in a community where the average availability is only 24%, we can achieve a minimum and average availability of 99.8% and 99.95%, respectively, at only 6X excess storage. With today’s low cost storage, this degree of replication seems quite feasible. This result demonstrates that such availability levels do not require sophisticated data structures, complex replication schemes, or excessive bandwidth. Indeed, our achieved availability levels are on par with today’s managed services [41, 65].

On the other hand, our results demonstrate that the presence of a small fraction of server-like members which are online most of the time is critical. A community where each member is available 33% of the time, giving an average availability of 33% (which is significantly more than 24%), requires 9X excess storage to achieve 99.9% availability even when the content and excess storage is uniformly distributed.
Chapter 5
Content search and ranking

5.1 Overview

In this chapter, we propose a novel approach to providing content based search and ranking across communities of several thousand nodes. Our approach consists of using an approximation of a state-of-the-art text-based search and rank algorithm \(^1\).

This chapter builds over the work presented previously on Chapter 3 and Chapter 4, by extending the multidimensional term-to-node index to support content search. First, we parse every hoarded file to extract keywords that are tracked using a local inverted index (i.e. a map from keywords to files). Then, all the keywords stored in the inverted index are advertised to other nodes using PlanetP’s global index.

Having indexed all the hoarded files, we support content search by simply using the two-level searching scheme presented on Chapter 3. In order to further constrain the search to only documents that are highly relevant to the query, we use the vector space ranking model \([107]\) and TFxIDF \([87]\). Briefly, the idea behind TFxIDF is to identify the relevant documents by looking at how frequent they use the query terms (TF) as well as how infrequent those terms appear across all documents (IDF).

We implement TFxIDF using only the information contained in the global directory as presented on Chapter 3. As shall be seen, for scalability it is important to limit the amount information used to rank documents, but this may reduce the effectiveness of TFxIDF. Furthermore, document replication can also reduce the effectiveness of TFxIDF as it affects the natural word frequency. Replication is common on communities like KaZaA and Gnutella where users hoard files for offline access \([44]\).

\(^1\)Our algorithm can be used to search and rank multi-media as well as text documents since today’s multi-media formats such as MP3 and AVI support the embedding of descriptive text.
Chapter 4 we use autonomous replication to improve file availability. In this chapter, we address several questions, including:

- How effective is PlanetP’s content search and rank algorithm given that it maintains a global index that contains even less information than previous related efforts [14, 40]?

- How does replication affects content search? In particular we look at the interaction between our autonomous replication algorithm presented on Chapter 4 and TFxIDF

In order to answer these questions, we use simulations driven by several existing documents collections. In particular, we first study our approximation of TFxIDF in the absence of replicas to show that PlanetP achieves search and rank accuracy that is comparable to a centralized solution. Next, we quantify the effect of introducing replicas on the document set.

### 5.2 Background: TFxIDF

In a vector space ranking model, data and queries are abstractly represented as vectors, where each dimension is associated with a different feature that describes the data. For example in the case of documents the value of each dimension is a weight representing the relevance of a particular term to a document or query. Given a query, we then compute the relevance of each document as the cosine of the angle between the two vectors using the following equation:

$$Sim(Q, D) = \frac{\sum_{t \in Q} w_{Q,t} \times w_{D,t}}{\sqrt{|Q| \times |D|}}$$

(5.1)

where $Q$ is the query, $D$ is a document, $|Q|$ and $|D|$ are the number of terms in $Q$ and $D$, respectively, $w_{Q,t}$ represents the weight of term $t$ for query $Q$, and $w_{D,t}$ the weight of term $t$ for document $D$. A similarity of 0 means that the document does not have any term in the query while a 1 means that the document contains every term in the query.
TFxIDF is a popular method for assigning term weights when dealing with documents. Note that if we were to index other type of information we would need a different technique that exploits its particular features. For textual information, TFxIDF combines the term frequency (TF) in a document with the inverse of how often that term shows up in the entire collection (IDF) to balance: (a) the fact that terms frequently used in a document are likely important to describe its meaning, and (b) terms that appear in many documents in a collection are not useful for differentiating between these documents.

There are several accepted ways of implementing TFxIDF [87]. In our work, we adopt the following system of equations from [107]:

\[
IDF_t = \log(1 + N_C/f_t)
\]

\[
w_{D,t} = 1 + \log(f_{D,t})
\]

\[
w_{Q,t} = IDF_t
\]

where \(N_C\) is the number of documents in the collection, \(f_t\) is the number of times that term \(t\) appears in the collection, and \(f_{D,t}\) is the number of times term \(t\) appears in document \(D\).

This leads to a similarity measure of

\[
Sim(Q,D) = \frac{\sum_{t \in Q} IDF_t \times (1 + \log(f_{D,t}))}{\sqrt{|Q|}}
\]  

(5.2)

where \(|Q|\) has been dropped from the denominator since it is constant for query \(Q\) across all documents.

5.3 Approximating TFxIDF

In designing PlanetP, we deliberately decided not to maintain the term frequencies and 
"\(t \to D\)" mappings necessary for TFxIDF in our global index to optimize space and reduce communication. In fact, with stop word removal and stemming\(^2\), our global index only contains the bare minimum of mappings from "important" words to peers.

\(^2\)Stop word removal eliminates words like "the", "of", etc.; stemming tries to reduce words to their root, e.g., "running" becomes "run."
We then approximate TFxIDF by breaking the ranking problem into two sub-problems: (1) ranking peers according to their likelihood of having relevant documents, and (2) deciding on the number of peers to contact and ranking the identified documents.

**Ranking Peers.** To rank peers, we introduce a measure called the *inverse peer frequency (IPF)*. For a term $t$, $IPF_t$ is computed as $\log(1 + N/N_t)$, where $N$ is number of peers in the community and $N_t$ is the number of peers that have one or more documents with term $t$ in it. Similar to IDF, the idea behind this metric is that a term that is present in the index of every peer is not useful for differentiating between the peers for a particular query. Unlike IDF, IPF can conveniently be computed using our constrained global index: $N$ is just the number of entries in the directory while $N_t$ is the number of “$t \rightarrow p$” entries in the global index.

Having defined IPF, we then rank peers using:

$$R_p(Q) = \sum_{\{t \in Q | (t \rightarrow p) \in I\}} IPF_t$$

which is a sum over all query terms contained in at least one document published by peer $p$, weighted by the usefulness of each term for differentiating between peers; $t$ is a term, $Q$ is the query, $I$ is the global index, and $R_p$ is the relevance of peer $p$ to $Q$. Intuitively, this scheme gives peers that contain all terms in a query the highest ranking. Peers that contain different subsets of terms are ranked according to the “differentiating potential” of the subsets.

**Selection.** As communities grow, it becomes infeasible to contact large subsets of peers for each query. To address this problem, we assume that the user specifies a limit $k$ on the number of potential documents that should be identified in response to a query $Q$. Then, given a pair $(Q, k)$, PlanetP does the following:

1. Rank peers for $Q$

2. Contact peers in groups of $m$ from top to bottom of the ranking\(^3\)

\(^3m\) represents a trade off between parallelism in contacting peers against potentially contacting some peers unnecessarily.
3. Each contacted peer returns a set of document URLs together with their relevance using equation 5.2 with IPF\(_t\) substituted for IDF\(_t\). This substitution is sufficient since peers maintain per-document term frequencies in their local indexes.

4. Stop contacting peers when the documents identified by \(p\) consecutive peers fail to contribute to the top \(k\) ranked documents.

The idea behind our algorithm is to get an initial set of \(k\) documents and then keep contacting peers only if there is a good chance of acquiring documents more relevant than the current \(k^{\text{th}}\)-ranked one. Simulation results show that \(p\) should be a function of the community size \(N\) and \(k\) as follows:

\[
p = C_0 + \lfloor C_1 N \rfloor + \lceil C_2 \sqrt{k} \rceil \quad (5.4)
\]

The tuple \((C_0, C_1, C_2) = (2, 1/300, 1/2.5)\) can serve as a good initial value for equation 5.4 since it works well for the benchmark collections studied in this paper (see Section 5.4). In general, we assume that users will adjust \(k\) when the results are not satisfactory (as they do when using Internet search engines). If users have to increase \(k\), then we should modify \((C_0, C_1, C_2)\) to increase \(p\). If users decrease \(k\) or never access the lowest ranked documents identified for queries, we should modify \((C_0, C_1, C_2)\) to decrease \(p\).

### 5.4 Performance

Having described PlanetP’s content search and ranking algorithm, we now turn to evaluating its performance. We start by assessing its efficacy in terms of finding relevant documents and ranking them appropriately. We then study the interaction between replication and content search to try to provide high availability for the global index’s information.

Our performance study is simulation-based but most of the parameters were derived from a prototype implementation. Also, we validated our simulator against measurements taken from the prototype when running up to several hundred peers.
<table>
<thead>
<tr>
<th>Collection</th>
<th>No. Queries</th>
<th>No. Docs</th>
<th>No. Unique Terms</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACM</td>
<td>64</td>
<td>3204</td>
<td>75493</td>
<td>2.1</td>
</tr>
<tr>
<td>MED</td>
<td>30</td>
<td>1033</td>
<td>83451</td>
<td>1.0</td>
</tr>
<tr>
<td>CRAN</td>
<td>225</td>
<td>1400</td>
<td>117718</td>
<td>1.6</td>
</tr>
<tr>
<td>CISI</td>
<td>112</td>
<td>1460</td>
<td>84957</td>
<td>2.4</td>
</tr>
<tr>
<td>AP89</td>
<td>97</td>
<td>84678</td>
<td>129603</td>
<td>266.0</td>
</tr>
</tbody>
</table>

Table 5.1: Characteristic of the collections used to evaluate PlanetP’s search and ranking capabilities.

### 5.4.1 Search Efficacy

We measure PlanetP’s search performance using two accepted information retrieval metrics, recall \( R \) and precision \( P \) [107]. \( R \) and \( P \) are defined as follows:

\[
R(Q) = \frac{\text{no. relevant docs. presented to the user}}{\text{total no. relevant docs. in collection}} \tag{5.5}
\]

\[
P(Q) = \frac{\text{no. relevant docs. presented to the user}}{\text{total no. docs. presented to the user}} \tag{5.6}
\]

where \( Q \) is the query posted by the user. \( R(Q) \) captures the fraction of relevant documents a search and retrieval algorithm is able to identify and present to the user. \( P(Q) \) describes how much irrelevant material the user may have to look through to find the relevant material. Ideally, one would like to retrieve all the relevant documents \( (R = 1) \) and not a single irrelevant one \( (P = 1) \). In our distributed context, it would also be ideal to contact as few peers as possible to achieve \( R = 1 \) and \( P = 1 \).

We assess PlanetP’s ranking efficacy by simulating and comparing its performance for five benchmark collections (Table 5.1) against a centralized TFxIDF implementation (called CENT). Each collection has a set of documents, a set of queries, and a binary mapping of whether a document \( D \) is relevant to a particular query \( Q \). Four of the collections, CACM, MED, CRAN, and CISI, were collected and used by Buckley [12]. These collections contain small fragments of text and summaries and so are relatively small in size. The last collection, AP89, was extracted from the TREC collection [48] and includes full articles from the Associated Press published in 1989.

We study PlanetP’s performance under two different documents-to-peers distributions: (a) Uniform, and (b) Weibull. We study a Uniform distribution as the worst case
for a distributed search and retrieval algorithm. The documents relevant to a query are likely spread across a large number of peers. The distributed search algorithm must find all these peers to achieve high recall and precision. The motivation for studying a Weibull distribution arises from measurements of current P2P file-sharing communities. Saroiu et al. found that 7% of the users in the Gnutella community share more files than all the rest together [88]. We have also studied a local community comprised of more than 1500 students sharing more than 10TB of data, which has a similar document distribution. Our Weibull distribution is parameterized to approximate the distribution found in this local community.

Figures 5.1, 5.2 and 5.3 plot average recall and precision over all provided queries as a function of $k$ for all the collections. Figure 5.4(a) plots PlanetP’s recall against community size with $k = 100$. Finally, Figure 5.4(b) plots the number of peers contacted against $k$. Like in the case of recall and precision, scalability behaves very similarly across collections. Therefore on Figure 5.4 we only show the results for the largest one (AP89).
Figure 5.2: Average (a) recall and (b) precision for the MED and CRAN collections when distributed across 100 peers. The legends $X.Y.Z$ are decoded as follows: $X = \{T: search\; engine\; using\; TF\times IDF,\; P: \text{PlanetP}\}$, $Y = \{W: \text{Weibull},\; U: \text{Uniform}\}$, and $Z = \{z: \text{indexed\; the\; most\; frequently\; appearing\; z\%\; of\; the\; unique\; terms\; in\; each\; document}\}$; for example, $T.W.100$ means TF\times IDF running on a Weibull distribution of documents, where all 100\% of the unique terms of each document was indexed.

We make several observations. First, PlanetP tracks the performance of the centralized implementation closely, even when we index only the most frequently appearing 30\% of the unique terms in each document. Further, PlanetP’s performance is independent of how the shared documents are distributed, achieving nearly the same performance for Uniform and Weibull. For a Weibull distribution of documents, when we index all 100\% of the unique terms, PlanetP’s recall and precision is within 11\% of CENT’s (average difference is 4\%). When we index only the 30\% most frequently appearing terms, PlanetP’s recall and precision is within 16\% of CENT’s, with an average difference of 14\%. These small differences demonstrate that it is possible to preserve TF\times IDF’s performance while limiting the global index to only a term-to-peer mapping. The good
Figure 5.3: Average (a) recall and (b) precision for the CACM and CISI collections when distributed across 100 peers. The legends X.Y.Z are decoded as follows: X = \{T: search engine using TFzIDF, P: PlanetP\}, Y = \{W: Weibull, U: Uniform\}, and Z = \{z: indexed the most frequently appearing z% of the unique terms in each document\}; for example, T.W.100 means TFzIDF running on a Weibull distribution of documents, where all 100% of the unique terms of each document was indexed.

performance given when we only index the top 30% of the unique terms indicate that we can further reduce the size of the global index at the expense of only a slight loss in ranking accuracy.

To further characterize the 4% average recall difference between PlanetP and CENT, we look at the document miss distribution. We define a miss as document that is both relevant and recalled by CENT, but is not recalled by PlanetP. We use the ranking position given by CENT to estimate the severity of the miss. Figure 5.5 shows cumulative number of misses against their ranking position for two different collections. We observe that at worst the misses are uniformly distributed, but there is a tendency to concentrate them on the low ranked documents (as seen in MED and less evident in
Figure 5.4: (a) Average recall as a function of community size for AP89 with \( k = 100 \). Variable and Constant \( N \) indicate whether the stopping condition adapts to the number of nodes \( N \) or not. (b) Average number of peers contacted in a community of 400 peers vs. \( k \) for AP89.

Second, PlanetP scales well, maintaining a relatively constant recall and precision for communities of up to 1000 peers. We have not study scalability beyond that point because the collections are not sufficiently large.

Third, PlanetP’s adaptive stopping heuristic is critical to its performance. Figure 5.4(a) shows that PlanetP’s recall would degrade with community size if the stopping heuristic were not a function of community size. (The effect is similar if the stopping heuristic was not a function of \( k \).) In addition, PlanetP’s adaptive stopping heuristic allows it to perform well independent of how the documents are distributed. Figure 5.4(b) shows that the dynamic stopping heuristic allows PlanetP to search more widely among peers when documents are more widely distributed, preserving recall and precision independent of document distribution.

PlanetP’s good distributed search and ranking performance does have a small cost: PlanetP contacts more peers than CENT. We observe from Figure 5.4(b) that while this cost is not trivial, it does not seem unreasonable. For example, PlanetP contacts only 30% more peers at \( k = 150 \) for the Weibull document distribution. Further, if we don’t use any ranking techniques and conduct a search by doing a boolean OR over the query terms we would have to contact on average 96% of the community (400 peers at
Figure 5.5: Cumulative number of misses according to their ranking position for AP89 (above) and MED (below). The total number of queries evaluated was 97 for AP89 and 30 for MED. The value of $k$ was selected to obtain 50% recall.

$k = 150$). On the same environment, PlanetP can retrieve the most relevant documents by just contacting 25% of the nodes.

5.4.2 Replication & Content Ranking

So far we have assumed that nodes are highly available and therefore queries can always achieve the maximum recall and precision possible for PlanetP. Using the replication algorithms shown on Chapter 4, we can also replicate indexing information to provide high recall and precision even under node failures. In this section we study how the introduction of artificial file replicas, affects ranking by changing the global term frequency (i.e. IPF).
Figure 5.6: *Distribution of the number of searchable fragments needed per file to achieve 3 nines for searching and ranking in CO and FS.*

First, in order to make the erasure coded fragments presented on Chapter 4 searchable, we need to embed some indexing information. Basically, a *searchable fragment* must carry the list of all the unique terms contained on the original file plus their frequency count. Using this information, nodes can advertise the content of the fragments in store by adding them to their local indexes. Searchable fragments introduce two types of overhead: they increase the fragment size and they take up extra space on the global index (i.e. the Bloom filters). Fortunately, the extra information added to a fragment can be highly compressed using the standard Unix gzip tool. Using a local document collection consistent of 449 PDF files, we observe that the average file size is 350KB and consequently the average fragment size would be 35KB. In order to make these fragments searchable we would need to add 5KB of indexing information (i.e. 14% overhead). Furthermore, the global index overhead can also be reduced by realizing that only one searchable fragment needs to be available in order to find and rank a file. Effectively, while files are fragmented, indexing information is replicated in full and therefore a smaller number of replicas is needed to guarantee the same availability as the file’s data. Therefore, we can use the equations introduced on Chapter 2.3 to decide how many searchable fragments have to be advertised in order to match the data’s availability.
To study the effect of replication over recall and precision, we extended PlanetP’s TFxIPF simulator to introduce searchable fragments. Using the FS and CO environments presented on Chapter 4, we derived the distribution of searchable fragments needed to obtain 3-nines availability for searching and ranking (shown on Figure 5.6). We don’t use the WG environment, because all nodes have the same availability and therefore files are homogeneously replicated. Note that only heterogeneous replication has the potential to interfere with IPF. The new simulator still implements the same ranking algorithm presented on Section 5.3, but before computing recall and precision all the duplicate results introduced by hitting on multiple searchable fragments from the same file are discarded.

On Figure 5.7(a) we use the MED collection to compare the performance of TFxIPF following the replica distribution of the FS and CO environments against having no replication. Similarly, Figure 5.7(b) shows the recall and precision degradation for CO when using the AP89 collection. Due to memory limitations on the simulator, the FS environment was not simulated with the AP89 collection.

On average Figure 5.7 shows a degradation of 1% when using replication. The reason for such small degradation is that the average number of artificial replicas introduced on the CO and FS environments (2.7 and 7.8 respectively) is small compared to the
average communal term frequency (i.e. 16 nodes per term). The little frequency change introduced by artificial replication is not enough to convert rare terms into popular terms and therefore it doesn’t affect TFxIPF based ranking. Observe that while the number of replicas needed to reach a certain availability target is not related to the community size, the average communal term frequency is likely to increase on larger communities. These trends suggest that highly available data and indexing information can be achieved without losing any performance on larger communities.

5.5 Summary

In this chapter, we have considered the problem of content search and ranking. Historically, good content search has meant a centralized system, where data is collected into one place for indexing and query processing. Here, we show how to do this in a distributed fashion.

In particular, we show how to provide decentralized content search, rank, and retrieval. Using PlanetP’s global directory, we robustly disseminate new information and replicate a limited amount of global state across the community. This information allows PlanetP to support a powerful content addressing model without requiring peers to maintain complex distributed data structures.

We have shown that PlanetP’s extremely compact global index does not affect its ranking accuracy: on average, PlanetP’s ranking performance is only a few percent less than that of a centralized implementation of TFxIDF. Further, the overall required storage is modest enough that our algorithms can easily scale to several thousand peers.
Chapter 6

Self managed federated services

6.1 Overview

In this chapter, we design a distributed resource management framework that can be used to build self-managing multi-site services that dynamically adapt to changing system configuration and client load. Our goal is to reduce the burden of deploying and managing a multi-site service to three tasks: (1) defining an application model for the framework to choose appropriate runtime configurations and specifying the desired availability and performance goals, (2) deciding the set of machines that can host instances of the service, and (3) providing maintenance support to repair these machines when they fail. Given a set of hosting machines, our framework will start an appropriate number of service replicas to meet the desired quality of service. Further, it will monitor the application as well as the hosting infrastructure; as the membership of the hosting set, processing capabilities and availability profiles of hosting machines, and client load change, our framework will adjust the number and placement of replicas to maintain the target quality of service.

Our management framework is novel in that it is completely distributed; each replica of a service is wrapped with a management agent that makes autonomous decisions about whether new replicas should be spawned and whether the current replica should be stopped or migrated to a better node. Agents rely on a set of loosely synchronized data to make their decisions, including the client load each replica is currently serving and load information about potential host machines. This autonomy and dependence only on loosely synchronized global state make our framework scalable and highly robust to the volatility inherent within federated systems. Despite autonomous actions based on weakly consistent data, we show that our management framework reaches a stable
and appropriate configuration rapidly in response to system dynamics.

As a proof of concept, we have implemented a prototype self-managing replicated UDDI service using our framework. This test application is challenging to manage because increasing the number of replicas also increases the overheads of writes into the replicated database. Thus, our manager cannot take the conservative approach of over-replicating to meet the desired quality of service. Also, as already stated, a UDDI service is a critical infrastructural service that can determine the availability of the overall federated system and so makes a good test case for our framework. Modifying this application to work with our framework was simply a matter of wrapping the back-end database with 270 lines of commented Java code.

We evaluate our prototype by studying its behavior in response to a simulated workload on two distinct testbeds. The first is a LAN-connected cluster environment that allows us to easily understand the behavior of our framework in a controlled environment. The second is the PlanetLab testbed [77], which consists of nodes widely distributed across the WAN. PlanetLab presents an extremely challenging environment for our framework because it is heavily loaded and very volatile in addition to being widely distributed. In fact, we expect that PlanetLab currently presents a more challenging environment than what would reasonably be expected of real federated systems hosting highly available services. Our results show that the self-management framework is quite stable despite its decentralized nature, yet adapts quickly and effectively in response to changes in load.

6.2 Self Management Approach and Algorithms

In our self-management approach, each self-managing application must have a model that maps possible configurations, i.e., number of replicas and their placement, to “fitness” values. We call this the application fitness function (or model). Each replica of a service is then wrapped with a management agent. Each management agent independently monitors the service and adapts the number of replicas and their placement as needed to meet overall performance and availability goals as defined by the fitness
function.

Broadly, our self-management algorithm works as follows:

- The set of nodes that can host services managed by our framework forms a PlanetP community, called the hosting community. Each node runs a monitoring agent that updates the node’s processing capacity, availability, and current load when there are significant changes (these are defined as properties of a node and so changes are propagated via the global directory).

- Each agent publishes the fact that a replica is running, and where it is running, when the replica (and the agent) is first started.

- Periodically, each agent searches for a better configuration using the application model and current information about the hosting community. If it finds a configuration with a fitness value that exceeds the current one by a threshold then it tries to deploy it. Before deploying a new configuration, however, the agent will wait a random delay period to avoid collisions. If, after the delay period, the agent observes that the service is still in the original configuration, then it proceeds with the deployment. Otherwise, it aborts the deployment.

Wrapping each replica of a service with a management agent is a fundamental decision of our framework. This decision implies that our framework must not limit the scalability of the service if it is to be applicable to a wide range of services. An alternative that avoids this coupling would have been to build a separate monitoring and management framework. We believe, however, that integrating the management framework with the application has two significant advantages. First, in the presence of failures, there will always be a management agent wherever a replica of the service is running. This means that, even when a network partition divides a service into several pieces, each piece can manage itself according to the semantics of the application, as defined by the fitness model, and the resource available. Second, our framework is unstructured and completely decentralized, making it highly robust even in very volatile environments.
In the remainder of this section, we discuss three critical components of our approach. First, we discuss the fitness model that we have developed for our example UDDI service; while this model was develop specifically for this application, the UDDI service is representative of a broad class of replicated applications to which this model may be applicable. Further, we explain the reasoning behind the model so that it can be applied to the construction of other models. Second, we describe the optimization algorithm that our framework uses to find good configurations. Finally, we describe how our decentralized decision-making algorithm works toward a coherent whole by choosing appropriate random delay times to avoid collisions.

6.2.1 An Example Application Model

We build our example model using availability and a single dimension of performance for simplicity. In particular, we focus on CPU-bound applications and build a model around CPU load and idleness. In general, for other applications, creating models based on other performance aspects, such as I/O and networking, would likely be necessary. We leave this as future work.

Each node’s availability is tracked as the average fraction of time that that node is a member of the hosting community. Each node’s processing capacity is estimated using a metric called bogomips. Bogomips is a portable metric used by the Linux kernel to assess any node’s processing capacity. Typically, any machine running Linux has an entry estimating its bogomips in the file /proc/cpuinfo. The processing capacity of a machine at any point in time is then defined as the estimated idleness of the CPU times the bogomips rating of the machine.

Given the above metrics for availability and performance and assuming that nodes’ availability are not correlated, we then estimate the expected processing capacity of a set of nodes, called a configuration, using the following equation:

\[
C(c) = \sum_{i=1}^{\lfloor c \rfloor} \binom{\lfloor c \rfloor}{i} A_{avg}^i (1 - A_{avg})^{\lfloor c \rfloor - i} i I_{avg}
\]  

(6.1)
where \( c \) is a configuration, \(|c|\) is the number of nodes in \( c \), \( A_{\text{avg}} \) is average availability of the nodes in \( c \), and \( I_{\text{avg}} \) is the weighted average of the current idle bogomips rating of all nodes in \( c \). \( I_{\text{avg}} \) is computed by weighing the idle bogomips of a node by its availability; this is necessary to prevent configurations with high performance but low availability machines from appearing overly attractive. Clearly, equation 6.1 is an approximation; in general, each node has a distinct availability and so an exact computation would require a sum across all possible combinations of nodes being up and down. This computation becomes expensive, however, as configurations grow in size and so motivates our approximation. In Chapter 4, we have seen that a similar approximation is generally conservative and works well for driving replication in highly heterogeneous P2P environments.

In essence, equation 6.1 gives the approximate expected capacity if the service is required to stay in the same configuration through node failures. This may be the case if, for example, the hosting community is under high utilization. If excess capacity is available, however, and the startup time of a new replica is less than nodes’ MTTRs, then equation 6.1 is pessimistic in that our framework will spawn additional replicas as needed to satisfy the offered load. This pessimism is desirable because, in general, we would prefer to exceed the availability target when resources are available rather than leaving resources idle and missing the availability target.

Given the above approach for estimating the capacity of a configuration, we then build a model that is generally suitable for a class of applications where:

- increasing the number of replicas increases overall throughput but can degrade the response time of a subset of the requests. Specific to the UDDI service, increasing the number of replicas will increase throughput because the workload is expected to be mostly reads, which only requires contacting one replica. The response time for writes will degrade, however, because writes must be applied to all replicas of the database.

- the service can only support a specific maximum number of replicas because of either administrative or scalability constraints. Specific to the UDDI service, the
replicated database that we use is experimental and so does not work well beyond a certain number of replicas.

- the number of replicas should be adjusted according to the current load, increasing when the load increases and decreasing when the load decreases. There are two reasons to reduce the number of replicas under low load. First, it reduces the response time of the operations sensitive to the number of replicas. Second, it releases resources so that other services on the hosting community can use them if needed.

Given the above properties, we derive the following rules that our fitness model should obey. As before, let $C(c)$ be the expected capacity of configuration $c$ as computed by equation 6.1, $l$ the current client load, $a$ the target availability level, $f(c, l, a)$ be the fitness of $c$ given $l$ and $a$, and $|c|$ be the number of nodes (equivalently replicas) in $c$, then for two different configurations $c_1$ and $c_2$:

1. if $C(c_1) \geq l \cdot a$ and $C(c_2) < l \cdot a$ then $f(c_1, l, a) > f(c_2, l, a)$;

2. if $C(c_1) \geq l \cdot a$ and $C(c_2) \geq l \cdot a$ but $|c_1| < |c_2|$ then $f(c_1, l, a) > f(c_2, l, a)$;

3. if $C(c_1), C(c_2) < l \cdot a$ or $C(c_1), C(c_2) \geq l \cdot a$ and $C(c_1) > C(c_2)$ then $f(c_1, l, a) > f(c_2, l, a)$;

4. if $C(c_1) < l \cdot a$, $C(c_2) < l \cdot a$ but $|c_1| > |c_2|$ then $f(c_1, l, a) > f(c_2, l, a)$.

One subtlety in the above rules is that, when computing $C(c)$ to compare against $l \cdot a$, we must cap the idleness of each node by $l$. This is because once all the client load has been directed to a single node because of failures, the excess capacity is no longer useful in serving that load. Thus, if the capped idleness gives a lower expected capacity than $l \cdot a$, then over time, this configuration cannot meet our expected load at the target availability.

Finally, we assume that a maximum capacity is specified for a service when it is run. Figure 6.1 shows one possible fitness function that fits our criteria, where $l$ is the current offered load, $a$ the target availability, and $l_{max}$ is the maximum expected offered load.
Figure 6.1: *Fitness function used for the UDDI service. \( l \) is the current offered load, \( a \) the target availability, and \( l_{\text{max}} \) is the maximum expected offered load. Each curve represents a different number of replicas.*

To compute the fitness of a configuration \( c \), we perform two computations. First, we compute \( C(c) \) where the idle capacity of each node is capped by \( l \). If this computation gives a bogomips value less than \( la \), then we return the corresponding fitness value. If this computation gives a bogomips value greater than \( la \), however, then we recompute \( C(c) \) without capping the idle capacity of each node and return the corresponding fitness value. This differentiates between configurations that can meet the current load at the target availability yet have different amount of excess capacity that can be devoted to respond to increases in client load.

### 6.2.2 Finding the Right Configuration

Given a fitness function, we must still solve the optimization problem of finding the optimal (or a close to optimal) configuration. We have chosen to use a genetic algorithm because: (1) as explained below, the manner in which this algorithm evolves toward a good configuration seems to fit our domain well, and (2) if needed in the future, the algorithm can be easily parallelized with minimal inter-replica communication.

A typical genetic algorithm consists of three steps [38]. First, generate an initial population—in our case, this can just be a set of random configurations. Second, randomly select a pair of individuals, where the probability of choosing particular individuals is weighted by their fitness according to some fitness function. Third, produce the
next generation by applying genetic operators to the selected pairs. Steps 2 and 3 are then repeated until a satisfactory offspring is found or a maximum number of generations has passed. In the latter case, the individual with the best fitness is chosen as the solution.

Genetic algorithms try to mimic the natural selection process by probabilistically allowing the most fit individuals to reproduce. The genetic operators most commonly used for reproduction are crossover and mutation. Crossover consists of taking a pair of individuals and crossing their chromosomes at a randomly selected point. For example, if the individual’s chromosomes were represented as bit arrays this would be a valid crossover $11001011 + 11011111 = 11001111$. The idea behind crossover is that the offspring of two genetically fit individuals can inherit the best attributes of both parents, becoming an even more fit individual. Using the same example, mutation would be represented as the flipping of random bits. The purpose of mutation is to prevent populations from becoming overly specific, limiting the selection process to some local optimum.

Genetic algorithms seem well suited to our application domain. Similar to our previous example, we can use bit vectors to represent service configurations where each bit indicates the placement of a replica at a particular node. Then, since a good configuration will likely include highly available, powerful nodes that are lightly loaded, a crossover of two good configurations is likely to produce another good configuration. Further, mutations prevent the search from collapsing around a local optimum.

In our current implementation, each management agent periodically searches for a better configuration according to its view of the community (provided through PlanetP). We use a publicly available library called JGAP [54] that provides standard tools for running genetic algorithms. Because of the simplicity of our fitness model, each search starts from scratch and runs for a fixed number of generations.

More complicated models might benefit in the future by starting from the previous population since it will likely shorten the search. Further, as already stated, genetic algorithms can be parallelized by randomly exchanging individuals between nodes. In our infrastructure, this can be done easily by gossiping promising individuals through
6.2.3 Realization of a Configuration

Because each manager agent in our framework operates autonomously, it is possible that several agents might try to deploy new configurations simultaneously. Concurrent deployments may interfere with each other, leading to too many or too few replicas. Particularly undesirable is when all agents conclude that the number of replicas should be reduced and simultaneously stop the local replica, leading to a service failure.

To address this problem, we adopt a randomized back off approach reminiscent of Ethernet-style back off \cite{66} to avoid transmission collisions. When an agent decides that a new configuration should be adopted, it waits a random delay time, \( t_d \), chosen with a uniform density function from the interval \([0, T)\), before deploying the new configuration. After \( t_d \), the agent rechecks the current configuration. If according to the global directory, the configuration has changed in any way, it cancels its intended deployment. Otherwise, it proceeds with its deployment.

In essence, our randomized back off approach provides a probabilistic serialization of actions across the agents. Multiple agents reacting to the same changes in the community use the random delays to avoid colliding with each other.

Given a fixed \( T \), what is the probability that two or more agents take concurrent actions if all of them simultaneously decide to deploy a new configuration? To answer this question, let \( N \) be the number of agents, \( v \) be the bound on information propagation time in PlanetP (also called the \textit{period of vulnerability} since members may have inconsistent views of the community during this time), and \( t_0 \) be the minimum chosen delay time. As explained in Chapter 3, \( v \) is only a probabilistic bound but all of our experimentations have shown this bound to be quite accurate except in very overloaded conditions. Since all agents choose delay times independently and each time within the interval \([0, T)\) is equally likely to be chosen, all outcomes are equally likely. Then, assuming that the delay times are discrete, we can compute the probability of concurrent action as follows:
Figure 6.2: Average delay plotted against the number of agents for a community of 200 nodes running PlanetP with a gossip interval of 1 second. The probability of collision was set to 0.1.

\[ p(N, T, v) = 1 - \frac{\text{NoCollision}(N, T, v)}{\text{AllOutcomes}(N, T, v)} \]  

(6.2)

where AllOutcomes is the set of all possible outcomes and NoCollision is the number of outcomes where \( \forall j, 1 \leq j \leq N - 1, t_j \geq t_0 + v \), such that all agents except the one choosing delay time \( t_0 \) will cancel their planned deployment.

The number of outcomes that satisfy the NoCollision condition can then be counted as:

\[ \text{NoCollision}(N, T, v) = N \sum_{i=0}^{T-v-1} (T - i - v)^{N-1} \]  

(6.3)

where \( T > v \). Basically, \( (T - i - v)^{N-1} \) is the number of possible non-colliding outcomes when \( t_0 = i \). The \( N \) factor arises from \( N \) possible agents that can choose \( t_0 \).

At runtime, we fix a probability of concurrent action, which is set to 0.1 in the experiments reported here, and vary \( T \) appropriately depending on the number of agents currently executing. Because Equation 6.2 does not have a closed form inverse, we have used the secant method to numerically compute \( T \) as the community changes.

To understand the delay that we might see in moving to a better configuration, we
computed the average delay before a redeployment is enacted. In essence, this bounds the responsiveness of our system to changes, either in the client load or the server infrastructure. Figure 6.2 plots the results vs. the number of agents for a community of 200 nodes running PlanetP with a gossip interval of 1 second. This gives a $v$ of 7 seconds. The probability of collision was set to 0.1. Observe that delays are under 2 minutes even for large numbers of replicas. Of course, delays heavily depend on the gossiping interval; if we back off of the somewhat aggressive interval of 1 second, the delays will increase.

Finally, observe that we also need to keep the redeployment time short to avoid collisions. The longer a node takes to instantiate a new configuration, the more likely that other nodes will start to concurrently deploy different configurations.

A possible solution to this problem is to modify the fitness function to favor new configurations that are not too different from the current one. The disadvantage of this approach is that could limit the search, leading to the selection of sub-optimal solutions. We instead introduce a deployment planning component to limit redeployment times without affecting the overall optimization algorithm. Specifically, our deployment planner will select a single step to enact per redeployment in order to bring the current configuration closer to the new one. The assumption is that if the new configuration is significantly better than the existing one then the optimizer will find it in every search and therefore the system as a whole will eventually get to the new configuration.

In order the find the best step, we first compare the number of server nodes on the existing configuration against the new one. If one is greater than the other then we do a linear search to find which single node addition or deletion will produce the best result according to the fitness function. If the two configurations have the same size then we search for the best migration (i.e. adding and then deleting a node).

### 6.3 Architecture

In this Section, we describe our prototype implementation. Figure 6.3 shows the architecture of our prototype in the context of our UDDI test application: each instance
of the service runs a multi-tier application comprised of the jUDDI [56] front-end, HSQLDB/R [50] (a replicated database) and PlanetP.

In this section, first briefly describe the runtime environments that we have developed for deploying PlanetP-enabled services. Then, we describe how we implement node monitoring, our self-management framework, and the UDDI service and client as PlanetP-enabled services.

### 6.3.1 Runtime Environments

Using ideas from Tomcat [96] and The Globus Toolkit [30], we package PlanetP-enabled applications inside jar files that include instructions on how they should be installed and executed. To run an application remotely, first a jar file is uploaded to the target node. Second, a loader service verifies that all the dependencies are fulfilled and downloads additional files if needed. Third, depending on the nature of the application a runtime environment is located and the binary code is handed to it. Optionally, an application can include a fitness model that will be handed to a local management agent for its execution.

Currently, we have implemented two different runtime environments. One is based on JWSDK [57] and Tomcat and is used to run Web Services like the jUDDI server. The other runtime, where most of our services run, is used for Java objects and is
implemented similarly to an EJB container. This runtime shares the same JVM with PlanetP so it allows services to use all the functionality provided by it.

One important extension to PlanetP that was developed in this work was the support for services. Services are Java objects that advertise themselves using active documents and that can be accessed over the network using RMI. Active documents are an extension of regular files that not only provide global naming, but are also responsible for handing stub classes to clients so that they can communicate with servers at runtime. Effectively, PlanetP and active documents work as a completely decentralized RMI registry.

A further extension to the RMI model was the inclusion of global file handlers. In order to be able to pass file handlers on function calls across nodes, we extended Java’s File class. This new abstraction provides the illusion of having read-only global files, which are very useful for example to share configuration files between management agents.

6.3.2 Node Monitoring

We have implemented a Node Monitoring service to provide load and idleness information for the evaluation of potential configurations. An instance of this service runs on each node of the hosting community and propagates static and dynamic information about the node (as node’s properties on the global directory). Examples of static information include machine type, the amount of memory, and the CPU capacity (in bogomips). Examples of dynamic information include applications currently running on the node and current idleness.

To reduce the overhead of propagating dynamic information, we only publish changes that are considered stable and exceed a certain threshold. For example, in the case of CPU idleness, we use exponential smoothing ($alpha = .5$) and only publish a new value when the change is stable and exceeds 20%.

6.3.3 Application Deployment and Management

Whenever a service containing a fitness model is started at a node, a new instance of the management agent is spawned. As already described, this agent is charged with
the management of the service. In addition to pursuing performance and availability goals, however, the management agent is also responsible for monitoring the health of the local replica and presenting a fail-stop failure model for the replica. For example, in the case of the UDDI service, if the local database stops responding then the agent will stop the entire service and undeploy all the dependencies so that the node can return to a familiar state.

6.3.4 The UDDI Service

As already mentioned, we build our test UDDI service using three open source components: jUDDI, HSQLDB/R, and JGroups. jUDDI is a UDDI front-end written as a Java web service that can run on Tomcat. Internally, jUDDI stores all registry information in a database accessed through JDBC.

Although the UDDI v3 specification includes a replication protocol, this feature has not been implemented in jUDDI. We instead replicate our service using a replicated database. In particular, we use HSQLDB, which is a full SQL compatible database written in Java. The authors of JGroups [55], an open source group communication infrastructure, have developed an experimental replicated version of HSQLDB (HSQLDB/R) using JGroups.

Our development of this service only consisted of wrapping the database into a PlanetP service, which required 270 lines of (commented) code. The fitness model for this service is as described in Section 6.2.1.

6.3.5 The UDDI Clients

To evaluate our framework, we have also written a client application that can impose load on the UDDI service. This client application itself is structured as a self-managing replicated service with a linearly increasing fitness function vs. the number of replicas up to a dynamically set maximum. Each replica generates a stream of requests with exponentially distributed inter-arrival times. These requests are load balanced across all the server replicas. In essence, this application really implements the proxies mentioned in Figure 6.3.
Client and service replicas are programmed to “repel” each other to avoid the need to carefully monitor the CPU usage of each application within the same JVM.

6.4 Evaluation

We now turn to evaluating the efficacy of our self-management framework. In particular, we study experimentally the responsiveness and stability of our management framework to changing client load as well as interfering external load and crashes of server nodes. We do not study the long term availability of the service because this would require running and loading the service experimentally for a long time\(^1\). We evaluate our framework using two distinct testbeds. The first is a LAN-connected cluster environment that allows us to easily understand the behavior of our framework in a controlled environment. The second is the PlanetLab testbed\(^2\), which consists of nodes widely distributed across the WAN. PlanetLab presents an extremely challenging environment for our framework because it is heavily loaded and very volatile in addition to being widely distributed. In fact, we expect that PlanetLab currently presents a more challenging environment than what would reasonably be expected of real federated systems hosting highly available services. Thus, results showing that our framework works well on PlanetLab are very encouraging.

6.4.1 Experimental Environment

Our cluster environment is comprised of 2 clusters, one with 22 dual 2.8GHz Pentium Xeon 2GB RAM PCs and the second with 22 2GHz Pentium 4 512MB RAM PCs. The two clusters are interconnected with 100Mb/s Ethernet LANs. All machines run Linux. Each Xeon machine is rated at 5570 bogomips and each Pentium 4 machine is rated at 3971 bogomips. Our experiments do not run sufficiently long so that meaningful

\(^{1}\) Alternatively, we could have studied this issue via simulation but decided against it because of the limited value of such a study. The only hypothesis that we would be able to validate is whether our approximation of a configuration’s availability is sufficiently accurate. Chapter 4 already gives us good reason to believe that this approximation works well, however its accuracy can always be increased, say by using several averages computed over subsets with similar availability instead of just one global average.

\(^{2}\) http://www.planet-lab.org/
availability measurements can be made for each machine. Thus, we arbitrarily set the availability of all nodes as reported by PlanetP to 90%.

PlanetLab is an open, globally distributed platform for developing, evaluating, and deploying planetary-scale network services. At writing time, PlanetLab has 336 nodes located throughout the globe. We run our experiments on 100 machines randomly selected from the set of .edu machines. We choose .edu machines because they are generally least restricted in terms of network ports being blocked by firewalls. PlanetLab nodes are highly heterogeneous, ranging in bogomips rating from 794 to 5570 bogomips. Again, we arbitrarily set the availability of all nodes to 90%.

As already noted, PlanetLab presents an extremely challenging environment. In fact, we had to turn off the replication of the database for the UDDI service because the volatility of the environment prevented JGroups from making any progress (similar to the case studied by Gupta et al. [46]). Thus, when running on PlanetLab, the UDDI service consists of independent replicas that can only service read requests.

The fitness model for the UDDI service is constrained with a maximum desired performance of 60000 bogomips and a maximum number of 4 replicas. In order to have room for the CPU demand to grow, the model also requires new configurations to have 20% more CPU capacity than needed.

We limit the maximum number of service replicas to 4 mainly because the experimental replicated database does not scale well beyond this size. The client application is also constrained to a maximum of 10 replicas. Turning off the replication, we have experimented with maximums of 8 service and 20 client replicas and the results are essentially the same as those presented here.

To emulate a realistic service, we populate our UDDI service with actual data obtained from http://www.xmethods.org, a site that lists publicly available web services. This site has over 3000 service providers registered to provide over 400 services. This gave us a database of approximately 3MB in size. Clients were written to issue random findBusiness queries against this set of registered providers.

During each experiment, nodes individually log information about themselves and replicas running on them. Each node periodically contacts a time server and logs timing
deviations together with the round trip latency. At the end of the experiment, we collect
the individual logs and merge them into a single ordered list of events. The accuracy
of this merge is on the order of two seconds. All reported data are averaged across 1
minute intervals to amortize inaccuracies introduced by skewed time lines.

6.4.2 Behavior on Cluster Testbed

Service Behavior. We begin by studying the adaptivity and stability of our framework
when the service is running on the cluster testbed. The hosting community is comprised
of all 44 machines in the two clusters. We start the experiment by deploying a single
instance of the client and one instance of the service on two random machines. Given
their respective fitness models, the client application quickly grows to 10 replicas while
the service stays at 1 replica in the absence of load. Once the client stabilizes at 10
replicas, we instruct it to progress through several different levels of load and observe
the behavior of the service.

The experiment has three phases. First, we slowly increase the load to observe the
server’s reaction. We start at 10rps and go up to 130rps in five steps taking a total of 200
minutes (time 0–200). At each step, a corresponding increase of 1 service replica on a
Xeon node would be sufficient to handle the offered load. Second, we abruptly decrease
the load from 130rps back to 10rps (time 200–250). Finally, we abruptly increase the
load from 10rps back to 130rps in a single step (time 250–350).

Figure 6.4(a) plots the behavior of the client and service while Figure 6.4(b) plots
system capacity, capacity allocated to the service, and capacity actually used by the
service. (While Figure 6.4 shows only one run of the experiment for clarity, we have
repeated this experiment many times with essentially the same results.)

Based on Figure 6.4, we conclude that our framework is quite stable despite its de-
centralized nature, yet adapts quickly and effectively in response to changes in load. In
particular, observe that throughout the experiment, the framework chooses configura-
tions that match the current load quite well, almost always using the minimum number
of required Xeon nodes. At each change in load during the first phase, our framework
starts a new replica within 1 minute. (Note that the actual length of the adjustment
Figure 6.4: *Experiments on the cluster testbed*: (a) Number of service replicas, number of client replicas, and client offered load plotted against experiment time. (b) Total system capacity, idle capacity, capacity allocated to the service, and capacity actually used by the service plotted against experiment time.

interval is dependent on a number of environmental settings such as the gossiping rate. Thus, we are less concerned with this magnitude as compared to whether the system makes the right decisions and its stability.) Even during large changes in load, our system responds smoothly although the adjustment time is of course longer. When load drops from 130rps to 10rps, our framework smoothly reduces the number of service replicas from 4 to 1 over 7 minutes. When load spikes from 10rps to 130rps, our framework again smoothly increases the number of service replicas back to 4 in just over 6 minutes. Note the longer adjust time of 6-7 minutes compared to the total adjust time of around 3 minutes in response to slow changes. This is caused by the fact that we limit the framework to one change per adjustment and wait at least 3 periods of vulnerability between adjustments (Section 6.2.3).

Of course, our framework is not perfect at all times due to its probabilistic nature and fluctuations in the observed load and system idleness. Between times 100 and 150, it twice deploys 5 replicas unnecessarily for short periods of time. Figure 6.4(b) shows that at these instances, the processing load shows peaks large enough to saturate the 3 replicas. Further, at each instance, two agents collided in their decisions to increase the number of replicas, resulting in the creation of 2 additional replicas instead of just 1. The agents quickly detect this over-replication, however, and stop 1 replica almost
Figure 6.5: Experiments on the cluster testbed: (a) Target load, completed requests, and failed requests plotted against experiment time. Note that at high loads, the clients were unable to reach the target load even when there are no failures because of inaccuracies in the Java timer; threads were often not woken up quickly enough to generate the desired rate of requests. (b) Average response time for completed requests with 95% confidence bars.

Interestingly, at time 155, 3 replicas of the service fail due to a buffer exhaustion error inside JGroups triggered by the high load. This failure is reflected in the reduced total capacity of the system because PlanetP fails the entire run-time system to enforce a fail-stop failure model, thus effectively removing the failed nodes from the hosting set. After the failure, the management agent on the remaining replica simply pushes the service back to 4 replicas in order to meet the offered load. Note that even if all 4 replicas fail, our framework would have restarted the service because we always run at least one additional agent separate from the application being managed.

**Impact of Self-Management on Clients.** We now consider how the above service adaptation to changing load impacts the clients. In particular, Figure 6.5 plots throughput, the rate of failed requests, and average response time as seen by the clients. Observe that response time can increase noticeably during adjustment periods because of warm up effects and temporary overload, particularly when more than 1 replica needs to be started. Very few requests actually fail during adjustment, however. A few tens of requests were lost around time 275 when the load rose sharply from 10rps to 130rps. In addition, a few requests failed when an excess replica (started because of a collision) was
stopped at time 106. Beyond this, all the lost requests were due to the service failure around time 155 and the service being loaded beyond its maximum capacity (as given by the cap of 4 replicas) around time 200.

**Response to Failures.** We have also studied the behavior of our framework under node failure (as opposed to the application failure in the above experiment) as well as interfering external load on nodes hosting the service replicas. We do not show the results here because they are entirely as expected. When a node fails, one of the remaining agents will eventually detect the failure and start a new replica when its PlanetP instance attempts to gossip to the failed node. We can speed up this detection by having the agent gossip heartbeats among themselves. When external load is placed on a node hosting a replica of the service so that it is overloaded, the agent quickly detects this overload and migrates the replica elsewhere.

### 6.4.3 Behavior on PlanetLab

Figures 6.6 and 6.7 show the results when we run a similar experiment on the PlanetLab testbed. In this experiment, we started loading the servers with 10rps, increasing to 130rps in five steps. As already noted, PlanetLab presents an extremely challenging environment for our self-management framework. In this environment, machines are so
Figure 6.7: *Experiments on the PlanetLab testbed: Target load, completed requests, and failed requests plotted against experiment time.*

heavily loaded that it can take up to several minutes for PlanetP to complete a gossiping round (i.e. synchronize the state of two machines). Even worst, TCP SYN packets are frequently denied by nodes with backed-up accept queues. At the application level, this condition translates to having frequent temporal inconsistencies in the membership directory.

The above problem is particularly evident for the clients since their fitness model considers all configurations with 10 nodes equal. Thus, replicas often land on heavily loaded nodes. For example between times 350 and 500, examining detailed logs revealed that the agents managing the client application were in frequent disagreement on the number of currently executing replicas. This is the cause of the instability for the client seen during this time period.

On the other hand, the UDDI servers are much more stable although they did make a number of adjustments between times 200 and 600, migrating replicas when hosting machines become overloaded and briefly reducing the number of replicas in the rare period of lower external load. The servers are more stable because their model rewards configurations containing less overloaded nodes. This allows PlanetP to more closely synchronize the shared global data. Note that because of the small amount of bogomips available per machine, the servers quickly reach the maximum number of replicas even when the client load is relatively low.
In summary, we find these results very encouraging. While there is much more instability compared to results from the cluster testbed, PlanetLab currently presents a very challenging environment. In fact, one would expect any practical environment hosting highly available services to be much less volatile than PlanetLab. Thus, the above results give us confident that our framework would work well in such realistic environments.

6.5 Summary

In this chapter, we have described the design, prototype implementation, and evaluation of a decentralized framework that can be used to build self-managing replicated services that dynamically adapt to changing system configuration and client load.

Given a set of hosting machines, our framework will start an appropriate number of service replicas to meet the desired quality of service. Further, it will monitor the application as well as the hosting infrastructure; as the membership of the hosting set, processing capabilities and availability profiles of hosting machines, and client load change, our framework will adjust the number and placement of replicas to maintain the target quality of service.

The decentralized nature of our framework makes it highly robust to the volatility inherent within federated systems. Our experimental evaluation on two distinct testbeds, one being PlanetLab open planetary-scale testbed, validates this robustness, showing that the framework adapts efficiently in response to changes but is quite stable even when operating under very challenging conditions.
Chapter 7

Applications

In this section we present two applications that are actively using PlanetP to deal primarily with membership management and information propagation. The first one is a P2P file systems called Wayfinder [75] and the other is a framework called Vivo [70] that is used for monitoring cluster based Internet services.

7.1 Wayfinder

As we argued on Chapter 1 locating relevant content on federated environments is becoming an increasing problem. Similarly, as data gets scattered across multiple nodes and users execute updates in-place, managing versions and propagating updates becomes a difficult task. This problem is further exacerbated as users increasingly depend on multiple devices such as PCs, laptops, and PDAs, some of which may frequently change in their connectivity and bandwidth status, requiring explicit reasoning about the impact of disconnected operation.

Wayfinder [75] is a novel P2P file system that advances the state of the art by unifying publishing, search and storage. In particular, Wayfinder addresses the problem of data management on federated systems by exporting three synergistic abstractions: a unified global namespace, content addressing, and availability-conscious replication.

Wayfinder runs as a federated service that constructs a universal shared namespace across the community. This shared namespace is formed by overlaying the local namespaces of participating nodes as shown in Figure 7.1. Each node’s local namespace is called its hoard and consists of a directory structure and files stored in a local file system. All the metadata created by Wayfinder to support this shared namespace is stored and propagated using PlanetP. Therefore, the community may at any point split into
Figure 7.1: Wayfinder’s shared namespace is constructed by merging local directory hierarchies (hoards) across connected nodes. This figure shows 5 nodes originally being connected so that the shared namespace is the merged view of hoards H1 through H5. When the connectivity changes and the community separates into three connected subsets, Wayfinder maintains a merged view for each subset. When the subsets reconnect, Wayfinder dynamically re-merges the shared namespace.

multiple connected subsets, each with its own shared namespace, and later rejoin to recreate the communal namespace. In essence, Wayfinder presents a shared view of all data stored across any set of connected nodes that expands and contracts smoothly on node arrival and departure.

Wayfinder supports content addressing by providing semantic directories [35, 39], which represent search queries and are populated with their query results. Semantic directories allow users to embed a multi-dimensional content-based navigation structure within the normal directory hierarchy. For example, a file discussing P2P file systems might be found through a semantic directory querying for “P2P” or through an other one querying for “file system”. More over, this structure actively organizes files as their content change since semantic directories are periodically reevaluated. In order to provide content addressing, Wayfinder parses shared files looking for keywords that are advertised through PlanetP using the algorithms presented on Chapter 2.4. Thanks to PlanetP search results can also be content-ranked to assist users in finding the most relevant files.

Finally, similar to our work on Chapter 2.3, Wayfinder explores the problem of file availability in particular to deal with disconnected operation. Although their availability
model is different, like in our algorithm they rely on PlanetP to track files and to propagate node availability.

7.2 Vivo

Vivo [70] is a research effort that focuses on understanding how to design cluster based Internet services for high availability. Their goals are to understand the relation between performance and availability in cluster based servers and to provide a systematic approach to improving availability.

Currently, they are working on understanding the impact of failures within Internet services due to their inherent complexity because of both scale and non-trivial dependencies among service components. In particular, they are building a self-representation of an Internet service that can be used at runtime to detect anomalies, diagnose failures and automate the validation of new system components.

Using PlanetP, they are building a framework where service components announce their presence to a monitoring agent. This agent constructs and monitors the service’s model to detect and correct any anomalies. Effectively, PlanetP provides membership management for components and works as a publish/subscribe channel matching components characteristics with subscriptions from potential monitoring agents.

PlanetP’s infrastructure allows the monitoring agents to deal with problems like disconnections, failures and rejoining of components by transforming them into simple high level events. Interestingly while the goal of PlanetP has been to serve on federated environments, its robustness has made it a good candidate for building fault-tolerant minded services on cluster environments.
Chapter 8
Conclusions and Future Work

In this dissertation we have studied the issue of building large scale federated systems. In particular we have used distributed probabilistic algorithms where members operate autonomously without depending on particular nodes. Our goal has been to build decentralized systems that can make progress in spite of faulty nodes, erroneous data, network partition, etc.

We started by motivating the disadvantages of building federated systems using a centralized approach together with traditional consistency algorithms to distribute data and control. Through the different chapters of this thesis we have shown over and over how critical infrastructural services can be implemented in a decentralized manner using only partial global information.

In Chapter 2.2 we use gossiping to robustly disseminate new information and to maintain loosely replicated state on federated environments. We show that changes to replicated data consistently reach all on-line members within several minutes. Further, synchronizing a global database, like the global directory, using our gossiping algorithm requires only a modest amount of bandwidth, even for extremely dynamic communities with very high rates of change.

Using this infrastructure, in Chapter 2.4 we build an extremely compact global index that we use to approximate TFxIDF. We show that limiting global information does not affect ranking accuracy: on average, PlanetP’s ranking performance is only a few percent less than that of a centralized implementation of TFxIDF. Further, the overall amount of storage and gossiping bandwidth used are modest enough that PlanetP can easily scale to several thousand peers.

In Chapter 4 we address the question of increasing shared data availability for a range
of distributed communities. We have attempted to quantify a conservative estimate of the amount of excess storage required to achieve a practical availability of 99.9% by studying a decentralized algorithm that only depends on a modest amount of loosely synchronized global state. Indeed, our results show that it is exceedingly difficult to achieve this level of availability if individuals do not at least have approximate knowledge of nodes’ availability and files’ current availability. The performance of our replication algorithm is extremely encouraging because it demonstrates that practical availability levels are achievable in a completely decentralized environment even with low individual node availability.

Our decentralized replication algorithm for heterogeneous communities incorporates a novel strategy for using erasure codes. This strategy allows nodes to increase file availability in an autonomous fashion without using complex data structures. We showed how to adjust the erasure encoding to accommodate nodes with widely different availabilities, yet at the same time achieve good overall data availability.

In Chapter 6 we describe the design, prototype implementation, and evaluation of a decentralized framework that can be used to build self-managing replicated services that dynamically adapt to changing system configuration and client load.

We have shown that our autonomous agents can effectively expand and contract the number of nodes running a federated service to accommodate load. Our approach not only helps reducing the burden on system operators but also provides recovery from even software bugs on the federated service itself.

To enable distributed coordination across agents, we designed an algorithm that provides distributed probabilistic serialization. This algorithm relies only on propagating the number of nodes which are likely to execute concurrent actions and broadcasting a single message to inform its outcome. We have provided all the analysis needed to probabilistically bound the chances of collisions and estimating the average time before seeing any action.

The decentralized nature of our framework makes it highly robust to the volatility inherent within federated systems. Our experimental evaluation on two distinct testbeds, one being the PlanetLab planetary-scale testbed, validates this robustness, showing that
the framework adapts efficiently in response to changes but is quite stable even when operating under very challenging conditions.

To summarize our work, we have provided substantial evidence to show that federated systems can be built using randomized algorithms. Their flexibility makes them ideal for a wide range of environments and resource constrains. As shown here randomized algorithms can be tuned to trade off resources for quality of service. In particular, we have studied this trade off on issues like fault tolerance, performance and consistency across several types services. In general we encounter that the storage and network requirements are sufficiently modest to warrant enough service improvement, when compared with existing solutions, for a variety of environments ranging from P2P to corporate networks.

8.1 Future Work

This thesis has shown that using probabilistic algorithms to build federated systems is a promising approach. However, we have just studied a few critical services that are common to any federated system.

Ultimately the goal is to build a framework that fulfills the role of a global operating system. This framework should capture the common abstractions that are used to build federated services. Through our work building PlanetP we have identified some of them, but more work building new services needs to be done to fully understand the problem.

For example, during the development of the global UDDI service we found that a global logging facility would simplify distributed debugging and data collection. The challenge of global logging is not only how to efficiently aggregate the communal output so that a programmer can use it, but also how to integrate the streams coming from multiple nodes into a common time line. In the case of the UDDI service we had nodes periodically synchronize their clock so that later we could get some reasonable bounds on the clocks’ deviation. A more efficient approach is the one presented by Cukier et al. [20] where nodes only piggy back information when they exchange messages, yet later they are able to reconstruct a global time line including worst case bounds for
each event.

The Gartner group has predicted that by year 2005, 10% of business interactions will occur via peer-to-peer-enabled platforms [33]. They say that services like nonlinear supply chains, multi-operated transactions, and multi-organization collaborative communities will push this vision forward. In order to accelerate the migration of business processes into federated environments we believe that methods for providing distributed auditing and accounting need to be studied. Furthermore, issues like access control and group management need to be extended to cope with multiple administration domains and decentralization in general.

Our self-management infrastructure is a concrete first step toward autonomous federated services. However, our fitness model is quite simple at this point in time. In order to support more complex services, compiler-based techniques need to be explored for assisting developers in designing fitness models. Similarly, models will have to include other performance and availability metrics than just expected CPU processing capacity. As models become more complex, our current approach of optimizing the service model at every replica may become too expensive for a single node to handle. Fortunately, genetic algorithms are well suited for cooperative optimization thanks to the idea of migrating individuals randomly from one node to another [15]. Future work may include coupling this idea with our existing gossiping algorithms. Furthermore, we may need to explore the approach of evolving existing solutions as opposed to starting from zero after each deployment.

While we did not start this work with the intention of scaling to millions or billions of users, we believe that it is possible to scale PlanetP beyond our initial target of five thousand peers if desired. One possible approach is to divide the community into a number of groups. Peers within the same group operate as described here. Peers from different groups will gossip an attenuated Bloom filter that is a summary of the global index for their groups. Peers mostly gossip within their groups but, occasionally, will gossip to peers from other groups. When searching, if the attenuated Bloom filter of group $g$ contains terms relevant to a query $Q$, then the searching peer, say $a$, would contact a random peer in group $g$, asking it to return a ranked list of peers in $g$ that
might have documents relevant to $Q$. A can then contact these peers using the current algorithm for ranking. Indeed, Gupta et al. [45] recently proposed using a hierarchy of peers in a very similar manner, although their system uses a distributed hash table across groups instead of gossiping.

Finally, note that while on chapter 4 we have focused on studying how to increase the availability of static content, our approach is not limited to the sharing of read-only data. At worst, updating a file is the same as injecting a new version and letting the replacement mechanism evict the old copies. Applications can greatly optimize this process, however, by propagating updates as diffs. Thus, we leave this as an issue for future work as update propagation and replication is highly application dependent.
References


Vita

Francisco M. Cuenca-Acuna

1994 - 1999  
Attended Universidad Nacional de Córdoba, Córdoba, Argentina.  
Major: Computer Science.

1999  
Bachelor of Science in Computer Science.

1999 - 2004  
Graduate Studies in Computer Science, Rutgers University, New Brunswick, New Jersey.

1999 - 2001  
Teaching Assistant.

2001  
M.S. in Computer Science.

2001 - 2004  
Graduate Assistant.

2004  
Ph.D. in Computer Science.

Selected Publications

2001  

2002  

2003  

2003  

2004  