Implementation and evaluation of an efficient, distributed replication algorithm in a real network

Submitted by
Luis de la Garza

Submitted on
March 31, 2010

Supervisor
Gerhard Weikum

Advisor
Mauro Sozio

Reviewers
Gerhard Weikum
Mauro Sozio
Eidesstattliche Erklärung
Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ........................................  ................................................
(Datum / Date)  (Unterschrift / Signature)
## Contents

1 Introduction 3
   1.1 Related Work .................................................. 6
   1.2 Contribution .................................................... 7

2 The Aeolus Testbed 8
   2.1 JXTA ............................................................. 8
   2.2 The Testbed ....................................................... 10
   2.3 Testbed’s Technical Considerations ............................. 12
      2.3.1 Structure of the Implementation .......................... 12
      2.3.2 Communicating Peers ....................................... 14

3 The P2R2 Distributed Algorithm 17
   3.1 The Item object ................................................. 22
   3.2 Message structure ............................................... 22
   3.3 Requests, special kind of messages ............................ 23
   3.4 Downloading items ............................................... 24
   3.5 Node component .................................................. 26

4 P2R2 within the Aeolus Testbed 29
   4.1 Server components ............................................... 30
      4.1.1 The NodeAdministratorService ........................... 31
      4.1.2 Topology management ....................................... 32
      4.1.3 Topology changes ........................................... 33
      4.1.4 Data collection from peers ................................ 35
      4.1.5 The DataConsumerService component ..................... 36
   4.2 Graphical display ................................................ 38
      4.2.1 Topology Overview Display ................................. 39
      4.2.2 Real Time Charts ........................................... 39
      4.2.3 Remote Errors Display ..................................... 39
      4.2.4 Node Status Display ....................................... 40
   4.3 Report Generation ................................................ 41
      4.3.1 Data Reports ................................................ 41
      4.3.2 Topology Reports .......................................... 42

5 Evaluation 42
   5.1 Parameters ....................................................... 42
   5.2 Measures ........................................................ 43
   5.3 Methodology ...................................................... 44
   5.4 Experiments and Results ....................................... 44
5.4.1 Variable number of items .......................... 44
5.4.2 Variable storage capacity of each peer .............. 46
5.4.3 Variable $\alpha$ parameter for the Zipf distribution .. 46
5.4.4 Variable number of churned peers .................... 48
5.4.5 Variable edge probability ........................... 49
5.4.6 Variable query ttl ................................... 53

6 Conclusions .............................................. 56
6.1 Future Work ............................................ 56
1 Introduction

Distributed Computing is not precisely new\cite{1}. Architectures such as Client-Server have been used to enhance computing power or to make use of remote resources.

Client-Server architectures rely on central computers (servers) providing services or resources to several other computers (clients). File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Hypertext Transfer Protocol (HTTP) and Web Services are examples of this architecture. Because remote, central server machines provide services or resources not present on the client computers, they become a single point of failure. This means that if the server machines undergo a failure or are disconnected from the network, clients will not be able to connect to them, thus, the provided resources won’t be available either.

Peer-to-Peer Architecture (p2p from now on) was popularized by file sharing systems in the late 90s. They have the natural advantage, such as Client-Server Architectures, of tapping into distributed remote resources. However, the reliability of p2p systems is higher, since they don’t rely on centralized services or resources to perform its tasks. This decentralized network poses many challenges, namely, resource availability (i.e., resources, such as data are contained in computers that might be unavailable or unreachable). Having unstable resource access brings down performance in p2p applications, even though there is no single point of failure. Given the nature of a p2p network, at any given time there might be different kinds of computers connected to it, this means that the computing power is not the same across the network, bringing interesting challenges.

Following the classification given in \cite{27}, p2p network architectures can be categorized as:

- Centralized. Contain a central resource or service on which the proper operation of the network depends. These architectures do not scale well and contain single points of failure. Napster is an example that illustrates this architecture (Napster’s centralized architecture was also the cause of its demise\cite{36}).

- Decentralized but structured. Even though these architectures don’t

\footnote{The Symposium of Principles of Distributed Computing, PODC, meets once a year since 1982}
have a central resource on which they depend, they are highly structured. They utilize a globally consistent protocol to make sure that any peer can route a search to some other node containing the desired resource. In order to guarantee this, these architectures impose a structured topology on top of the physical network topology (i.e., overlay topology). The Freenet Project\cite{37} is an example of such an architecture.

- Decentralized and unstructured. These are networks where there exists neither centralized resources nor a structure in the topology or in resource placement. In order to find resources in the p2p network, peers must query their neighbors. We are particularly interested in this kind of architectures. Gnutella\cite{23} is an example of such an architecture.

Several distributed systems are currently available in order to assist the execution of diverse tasks, from resource sharing to data indexing. The following is a list of a few examples of such systems:

- Grid File Systems\cite{2}[?] benefit from several small storage areas (e.g., hard drives) distributed over a network to enhance availability. Maintaining a high availability in such a distributed file system is a great challenge; a single File Table\footnote{The File Table keeps track of the location, or multiple locations of files and its blocks} must be maintained in order to manage each file and the blocks that compose those files. Grid File System ensures high availability by means of using data replication among the connected peers.

- Distributed Search Engines\cite{3}\cite{4}\cite{5} exploit the computing power of peers that belong to a p2p network in order to index the Internet (or any information system) via web crawling.

- File Sharing Networks\cite{22}\cite{24}\cite{25} rely on a vast number of users (peers), each sharing files and at the same time interested in obtaining files from other peers. Early versions of file sharing networks downloaded items from only one user, while newer implementations support simultaneous download of a file from several peers.

- Grid Computing\cite{38}\cite{39} aims to combine several computer resources that belong to multiple entities for a common goal. The main challenge of these systems is that each computer can potentially belong to a different administrative domain. This means that each peer is subject to different network security restrictions, therefore, a sort of middleware must be provided to access the resources in a uniform way.
The main challenge for p2p applications in unstructured networks is how to perform data replication. Without data replication, the performance of the network depends solely on the availability of items (which in turn depends on the availability of peers). With data replication, it is possible to make copies of items and distribute them across the unstructured network, in order to improve the availability of items, increasing the performance of the network. Obviously, generating replicas of all items would be not only inefficient but also not feasible, since in real data networks, the number of items in the network is much larger than the number of peers connected, posing storage difficulties. An optimal replication algorithm generates the right amount of replicas of the right items. To show the advantages of data replication an implementation of a data replication algorithm (the P2R2 Distributed Algorithm[6]) will be presented.

As real networks go, real network problems go. Any reliable p2p application must handle unexpected scenarios (e.g., connection resets, peers unavailable, lost messages and so on) that are not under the application’s control and should not expect peers to behave in a stable or reliable way, much less assume that all sent messages reach their destination. As such, the presented implementation includes extensive error handling routines and robust error recovery procedures. We have also dedicated considerable effort to construct a framework to assist in the execution of controlled experiments, as well as data collection from remote peers.

Throughout this paper, we will focus on the Implementation and Evaluation of a Data Replication Algorithm in an Unstructured Peer-to-Peer Data Network with Read-Only Files. In our implementation, each peer contains data items and is interested in obtaining some other data items from other remote peers. Peers in these networks communicate via query forwarding. The performance can be easily measured as success rate (i.e., satisfied queries to sent queries ratio).

In order to get access to a heterogeneous set of computers, we used the Aeolus Testbed[13][12] to construct, test and execute our implementation. The Aeolus Testbed is a distributed system of large scale that supports the development, execution and testing of applications written in Java using JXTA[9] and the Testbed’s API. It manages a set of computers geographically distant, abstracts connection details and offers developers an easy to use API to tap into the resources of these remote computers.
1.1 Related Work

Cohen and Shenker [26] lay down a theoretical model of the data replication problem. They provided proof that the most obvious replication strategies (Uniform and Proportional) are not only outperformed by all strategies but also achieve identical performance for soluble queries\(^3\). Furthermore, the theoretical optimal data replication strategy for systems with soluble queries was identified: Square-Root Allocation strategy.

Given the unstructured nature of the network, the global query rate of each item is unknown to all peers, so, having a distributed algorithm using a Square-Root Allocation strategy is a non trivial and hard task. Also, questions such as location of replicas and obtaining replicas is not made clear (in their model, Cohen and Shenker [26], the term copies refers to actual copies or data pointers), so the actual obtainment of files is not treated.

Knowing the theoretical upper bound of a data replication algorithm is of utter importance. Nevertheless, implementation and design issues such as finding the right search method are also very important and play an important role. On a less theoretical note, yet still not using full-fledged implementations,[27] includes an analysis of various search methods (namely, expanding ring, k-walker random walks and flooding) through the use of simulations on topologies based on real p2p networks. Of special interest is that popular protocols such as Gnutella[23] utilize flooding search methods, whose major limitation is overloading each peer with potentially duplicate messages and does not escalate well. As it will be shown, the P2R2 Distributed Algorithm achieves near-optimal performance with a very low footprint due to the fact that it uses simple 1-walker random walks as a search method, without the state and check approaches mentioned in [27], thus avoiding extra code in order to keep track of walkers. An important aspect of using 1-walker random walks is that the number of messages needed for each query is very low compared to other schemes.

Another interesting data replication algorithm is the Pull-Then-Push replication[28]. It is very powerful and very simple (it achieves Square-Root Allocation). The way [28] presents the replication algorithm is by defining two phases: 1) when peers search a data item, they are in the pull phase, 2) after a successful search, the requesting node enters the push phase, transmitting the data item to other nodes to generate replicas. It is noteworthy that

---

\(^3\)A soluble query is a query issued for an item that exists on the network. Querying for items that don’t exist on the network are outside the scope of this document.
in Pull-Then-Push, all peers that receive the push message create a replica of the item. In a real network, this means that the item has to be pushed to several nodes, therefore, this might not be very a very efficient use of the bandwidth, due to the inherent size of a typical shared file in commonly used file sharing systems[35].

Contrary to the principles on Pull-Then-Push[28], [29] proposes a Local Minima Search (LMS) algorithm in which the owner of each item takes the responsibility of placing replicas of the item on several nodes. A local minimum for an item is defined as the node, $u$, whose ID is the closest to the item’s ID in $u$’s neighborhood$^4$. The main advantage of this approach is that several replicas for an item will be available throughout the network, which leads to an increased performance. However, this can also lead to scenarios where too many replicas for items exist in the system, and it might be that the combined storage capacity of the peers would not suffice to produce the desired success rate.

Projects such as Andrew File System (AFS)[30][31], Carbonite Replication Algorithm [33], DHash++[34] and the Total Recall peer-to-peer Storage System[32] are just a few examples where data replication has been brought into a project to increase reliability and availability of a system.

1.2 Contribution

Our main purpose is not to introduce a novel data replication algorithm, rather, to present an implementation of the P2R2 Distributed Algorithm [6] and the construction of a system to evaluate its performance under several conditions as well as a systematic methodology to evaluate it.

Even though we took an already existent algorithm, the presented testing methodology and the design that was used for the implementation can clearly benefit other data replication algorithms or p2p network systems. Testing data replication algorithms on real networks is very important, and our implementation makes the testing of other algorithms simple by ”plugging in” algorithm code. We also offer an intuitive understanding of the main factors that can contribute to or detract from the P2R2 Distributed Algorithm performance by testing the effects of several parameters.

We identified, in due time, several pitfalls when it comes to implement a

$^4$Nodes within $h$ hops of $u$ in the network, as the authors define it
p2p network and evaluating it, offering insights on their causes and a viable solution that we proceeded to implement. These insights can benefit potential p2p systems designers, implementers and testers, so they can avoid them in a timely manner.

Being able to perform repeatable experiments in an easy way is a main factor in any scientific evaluation. Part of our contribution is a framework that can be used not only to collect data from remote peers, but also to manage peers in order to execute experiments with rich and usable results.

2 The Aeolus Testbed

In order to simplify the clarification of concepts, throughout the rest of this document, we will assume the following notations:

- objectName.propertyName. Refers to a property (attribute) named propertyName that exists in an object whose name is objectName.

- objectName.methodName(argumentsList). Refers to the invocation of a method named methodName that exists in an object whose name is objectName and that takes a list of arguments argumentsList.

We chose UML as the language used in our diagrams. UML is easy to understand, has a wide power of expression and there are many publications containing information about it.

2.1 JXTA

The Aeolus Testbed API relies on JXTA’s API. JXTA is a set of open, generalized p2p protocols that allow any networked device, such as sensors, cell phones, PDAs, laptops, workstations, servers and supercomputers, to communicate and collaborate mutually as peers. Following [15], we offer a brief summary of what JXTA is.

Common tasks in p2p programming such as peer discovery, resource advertisement and discovery, peer monitoring, organization of peers into groups and communication with other peers are standardized by JXTA by using protocols. JXTA protocols are designed to provide programming language and underlying transport protocols independence. As such, they can be implemented in several programming languages (although we used a Java
implementation[16]) and over different network transport protocols (such as Bluetooth and TCP/IP).

The JXTA network is composed by connected nodes or peers. Peers can also belong to a peer group, if there is such a need. A JXTA peer is basically any device providing an implementation of JXTA core services. Peers might also provide additional standard services and are connected by any suitable networking protocol. Each peer contains resources or services that are made available to other remote peers. Services and resources are application specific items that provide certain functionality, such as databases, access to specific devices such as radio telescopes or interactive programs. Similar to peers grouping themselves into groups, there are peer services and group services. Peer services are provided by a single peer, while group services are provided in either a federated, redundant or cooperative way by the peer group.

JXTA peers advertise their resources and services by using advertisements, which are simply XML documents containing information about the offered resources, such as ID, a simple description and so on. Advertisements allow peers to discover resources and to determine how to connect and interact with those resources. Based on their role in the JXTA network, JXTA peers can be categorized into three types:

- Minimal Edge Peer. Peers implementing only the required core JXTA services. Simple devices, such as sensors are typical minimal edge peers. These peers require the assistance of other peers (proxy peers, see below) to be able to fully participate in a JXTA network.

- Full Edge Peer. Peers that implement all the core and standard JXTA services, thus, being able to participate in all of the JXTA protocols.

- Super Peers. Super peers are peers with specific functions that implement and provide resources to support the operation of a JXTA network, these functions are known as superpeer functions. There are three super peer functions:
  - Relay. Stores and forwards messages between peers that do not have direct connectivity due to firewalls or NAT.
  - Rendezvous. Maintains global advertisement indexes and assists with advertisement searches and message broadcasting.
  - Proxy. Designed in order for minimal edge peers to get access to the JXTA network functionalities.
In order to maintain communication among peers at the application level, JXTA provides *sockets* and *pipes* objects. Sockets are similar to regular Sockets in the way that they are reliable bi-directional connections. Pipes, on the other hand, are an asynchronous, unidirectional message passing mechanism (conceptually similar to Datagrams). JXTA Sockets are a specialization of the well known Java™ 1.6 Socket, which makes JXTA Socket’s API simple to use and understand.

### 2.2 The Testbed

In order to test the implementation in a real network scenario, we were granted access to the Aeolus Testbed[13][12]. With it, we were able to run the implementation’s code in several computers geographically distant.

The Aeolus Testbed relies on JXTA to provide connectivity between the computers that compose it. It provides an easy to use Web Interface where code can be uploaded, monitored and managed. One of the great advantages of the Testbed is that it provides access to a large collection of heterogeneous computers and, as such, they provide enough instances where exceptional situations arise (e.g., computers leaving abruptly the network, computers running other processes at the time of testing, or reset connections), forcing the code comprising the implementation to be really robust. The Testbed allows to run *virtual instances* on computers, this means that a computer can run several instances of the code. Since other research groups also work on the Aeolus Testbed concurrently, it is possible to independently assign disjoint sets of computers. In Aeolus Testbed terminology, each disjoint set of computers assigned to a user receives the name of *allocations*.

Details such as IP addresses, port numbers, host names, firewalls and so on are well hidden behind JXTA. JXTA is a framework designed to *create peer-to-peer applications based on Java technology*. With those principles in mind, our implementation is freed from basic network programming details. The Testbed, along with JXTA, provides an easy to use interface in order to "plug in" one’s code to the Testbed’s network.

Even though JXTA and Aeolus Testbed solve common basic network programming problems, the discussed implementation is not free of unexpected scenarios. Given the nature of the Aeolus Testbed network, the implementation must handle a vast array of scenarios that can possibly affect the performance.
Research groups, universities, institutes and other participants in the Aeolus Project can connect computers to the Aeolus Testbed by installing the Testbed’s software[14] on these computers (see figure 1). Once these computers are connected to the Testbed, it is possible for any participant to use them in their allocations.

An allocation is a set of computers connected to the Testbed, each running up to ten virtual instances (see figures 2 and 3). A computer can belong only to one allocation. Once an allocation has been created, it is possible to upload configuration files, libraries and code and decide which computers from the allocation should have access to which files. It is also possible to decide which computers from the allocation should execute which services.

In order to actually perform some tasks in the Aeolus Testbed, it is needed from the developers to create services. The Aeolus Testbed can only execute code included inside services. A service is a simple Java class that implements the `aeolus.edge.Service` interface from the Testbed’s API. Once a service has been created, it can be registered to be used by any allocation.

A very important feature of the Aeolus Testbed is that it makes allocations to be independent of each other and makes sure that peers in an allocation can be discovered by other peers in the same allocation. This implies that, even though each computer in an allocation might be part of a different domain, Aeolus makes sure that they can be connected; also, Aeolus keeps allocations independent even if different allocations contain computers in the same domain, so, services, resources (and even problems) will be isolated to one allocation.

Whenever an allocation is created, Aeolus Testbed assigns a JXTA rendezvous peer to it, so all computers in the same allocation will use the same rendezvous peer. Aeolus makes sure that different allocations will have different JXTA rendezvous peers. As it was discussed in the previous section, JXTA rendezvous peers maintain global advertisement indexes, and by doing that, they help remote peers behind firewalls to discover each other. Through the use of JXTA rendezvous peers, the Aeolus Testbed handles connectivity problem providing a common connection point for all computers in an allocation (see figure 4) and makes sure that allocations are kept independent.

The work flow to get code uploaded and executed in the Aeolus Testbed

---

\[5\text{ Services can be made public or private}\]
requires several steps through a graphical web user interface. After the code has been successfully compiled, it must be uploaded (in the form of a jar file) to an allocation through the Aeolus Testbed’s web site, along with any needed configuration files. Upon completion, the allocation must be restarted for any changes to be reflected. Through visual feedback from the web site, it is possible to monitor the status of the restart process, and once done, the registered services in the allocation are started. Since there are many detailed steps in this process, it is very error prone, consequently, we had the need to automate this process by means of our own supporting scripts and ant interfacing tasks.

![Figure 1: Participants can connect computers to the Testbed](image)

### 2.3 Testbed’s Technical Considerations

#### 2.3.1 Structure of the Implementation

As it is required by the Aeolus Testbed, our implementation was developed using Java\textsuperscript{TM} 1.6 and JXTA\textsuperscript{[9]}. We also benefited from free open source libraries \[7\] [8] [9] [10] [11] to perform and/or automate several tasks.
The computers utilized to perform our experiments belonged to the Aeolus Testbed. Due to still unknown reasons, we could not always work with the same set of computers to perform our tests. However, plugging computers temporarily to the Aeolus Testbed is fairly simple and sometimes it was necessary to include our own extra computers from our department to perform meaningful experiments.

The distributed nature of the implementation has inherent complications when it comes to collect data in a centralized way. To that effect, it was needed to clearly divide the work for the following tasks: implementation of the P2R2 Distributed Algorithm, manage experiments, collect data from remote peers and process collected data (more on this in section 4). The implementation of the P2R2 Distributed Algorithm resides solely in the Testbed’s computers, while support tasks such as peer administration, data collection, statistics, report generation and data display reside mainly in the so-called Server Components. At this point, it must be clearly stated that the single reason these server-like, centralized components exist is to assist in tasks such as collect experiment data to measure performance, manage peer connections to study special topologies and send messages to perform controlled experiments. In no way we implemented a centralized data replication algorithm.

Our implementation is highly configurable via property file. Since the only access to the Aeolus Testbed available for developers is through a web site, it was not possible to provide our implementation with command line arguments (which makes script aided testing simpler). However, we developed
a series of scripts and **ant** build XML documents to go around this limitation in order to modify the configuration file with the desired experiment parameters before uploading it to the Testbed and executing an experiment.

### 2.3.2 Communicating Peers

One of the main challenges in developing applications that work over a network is making sure messages arrive to their destination. Our implementation uses serialized objects in order to simplify message transfer. Even though JXTA is reliable, sometimes messages get lost. This is why the implementation includes code that makes its best effort in order to assure that
sent messages are received. By means of special acknowledge messages, con-
trol codes, keeping track of messages that could not be sent due to failures
in order to send them later and ignoring duplicate messages that (due to
inherent network race conditions) are sent more than once, the number of
lost messages is tried to be kept to a minimum.

The Aeolus Testbed defines two basic concepts: Service and Disco-
veryService. A Service is a piece of code containing, among several other
attributes, an ID and offers some functionality to other computers in the
Aeolus Testbed network. The Aeolus Testbed publishes information about
services, such as its ID, so other peers can discover it. When a peer is search-
ing for other peers, it uses a DiscoveryService, providing adequate call-back
methods that will be invoked upon discovery, so, when a Service has been
found in the Aeolus Testbed network, these call-backs are executed and peers
can start to use services. After discovery, all communication boils down to
use instances of JxtaSocket (which, as discussed earlier, is a specialization
of the well known Java™1.6 Socket). Once this JxtaSocket is obtained, it
is passed to a Connection object to be managed.

In order to decouple peers from JxtaSocket or Aeolus Testbed specifics,
the Connection object, upon instantiation, requires a message queue and
an endpoint (identified by a valid peer ID). This message queue is the only
point of interaction between peers and Connection objects, following the
producer/consumer paradigm. Connection instances can be set as writing or
reading connections. Writing connections will consume any message avail-
able on the message queue and proceed to send it to the assigned endpoint
through a JxtaSocket, while reading connections will read messages from
the endpoint (from the JxtaSocket, to be specific) and deposit them (i.e.,
produce) into the message queue. The only action required from peers to use
Connection objects is to deposit/take messages into/from a message queue.
This decoupling action is not only practical when using a Connection, but
also it is of great help to test each component individually. Figure 5 illus-
trates the relationship between peers and Connection objects.

It is interesting to note that peers can only send and receive objects that
implement the PeerMessage interface; this design decision was taken in order
to be able to expand the range of messages that can be interchanged between
peers without modifying Connection code. It also is a good practice to sep-
arate the logic of how to send a message and how to interpret a message. For
a more detailed discussion on messages, please refer to section 3.2.
Figure 5: Relationship between peers and Connection objects

Having incoming and outgoing connections (reading and writing connections, respectively) seems to be counterintuitive. We assume that if a peer can receive messages from a remote peer, then it can also send messages to it. Since peers have no control when a message will be received, connections that are reading messages would be constantly blocked waiting for incoming messages. At first sight, this seems that the time spent waiting for incoming messages (i.e., reading from the JxtaSocket’s InputStream) could be used to send messages (i.e., write to the JxtaSocket’s OutputStream). This does not seem like a problem, due to the fact that it is safe to use both streams concurrently (we, of course, would need a dedicated thread to write messages and another one to read messages). One must have in mind that it is of great interest of each Connection to guarantee that each message arrives in its entirety to its destination. A common and very simple way to assure safe message arrival is using acknowledge messages. Thus, every time a Connection receives a message, it sends back an ack code to the remote Connection that delivered this message; likewise, every time a message is sent, an ack code is waited for some time before sending the next message). Using acknowledge messages means that either writing or reading messages imply using both JxtaSocket’s streams. We have stated early that reading messages is a blocking operation, therefore, if we used only one Connection per pair of connected peers, the rate at which peers write messages will decrease. Figures 6, 7, 8 and 9 illustrate this.
A very important aspect of Connection objects is that they have the ability to regenerate the underlying JxtaSocket if needed to. They also expose a method so users of Connection objects can close the connection at any time, without blocking. This means that if an unrecoverable error when sending/receiving a message is detected, the JxtaSocket will be closed (thus, suspending temporarily the transfer of further messages), a new one will be created and transfer of messages will be resumed. In the meantime, any peer using this Connection is not affected, since, as we have stressed before, the only communication between peers and their connections is through the message queue. Even though this is a concept very simple to understand, it involves interleaved complex synchronization code to make this process defect free.

3 The P2R2 Distributed Algorithm

Following the P2R2 Distributed Algorithm presented in [6], we implemented an \( \alpha \) approximation algorithm of the knapsack problem (refer to algorithm 3). The procedure receives a set of items that a peer \( p \) is aware of, along with the score for each one of them and returns a set containing the items that peer \( p \) should get a replica of. For presentation purposes, we assume that the \( \text{itemsAwareOf} \) collection contains a way to iterate through all items, from beginning to end.

The most CPU intensive part of this algorithm is the sorting of the \( \text{itemsAwareOf} \) collection (we confirmed this in the early stages of development by using a profiler[18]). We rely on the already included sort method in
Figure 7: Dedicating one connection to read

the Java™1.6 development environment, which has a guaranteed $O(n \log(n))$ running time, giving our knapsack implementation the same running time. The rest of the P2R2 Distributed Algorithm is as presented on listing 2.

The purpose of the P2R2 Distributed Algorithm is to determine, for each peer, the set of items that shall be obtained as replicas. The algorithm does this in a near optimal fashion, this means that the amount of replicas made across the network is close to the optimal amount.

The algorithm mandates to generate replicas for items whose score is greater; this score is a numerical evaluation based on the item’s $r_i$ counter and its size, as stated in line 2 of our knapsack implementation. Every time a peer receives a request for item $i$, the counter $r_i$ might get incremented (line 26 of algorithm 2). Clearly, the greater the value of this $r_i$ counter for item $i$ in peer $p$, the more probable it is that a replica will be placed in $p$.

Clearly, in order to generate the near optimal amount of replicas for an item $i$ across the network, the value of $r_i$ for different peers must be different (since item’s size remains constant). Inspecting closely line 26 for the P2R2 Distributed Algorithm, it can be seen that the value $r_i$ for item $i$ is incremented if the received request has not been satisfied or if the id field of the
peer that satisfied the request is lexicographically less than the id of the peer receiving the request. A peer will satisfy a request if it contains an item; if this item is in the untouchable set of items, then the field satisfiedBy will contain an empty string \(^6\). The lexicographical order we use\(^7\) guarantees that an empty string will always have a lesser lexicographical value than any other string. The algorithm requires that the peers have unique id’s, which is something our system provides by using the Internet Address of a peer.

\(^6\)A string whose length is zero. Not to be confused with null
\(^7\)Java™ 1.6’s java.lang.String.compareTo() method

**Algorithm 1** knapsack(capacity, itemsAwareOf, \(r_0, r_1, \ldots, r_i\))

1: \(t \leftarrow 0, \text{itemsToObtain} \leftarrow \emptyset\)
2: sort itemsAwareOf in descending order according to \(\frac{r_i}{i \cdot \text{size}}\)
3: while \(t < \text{capacity}\) and itemsAwareOf.hasMoreItems() = true do
4: \(i \leftarrow \text{itemsAwareOf.getNext}()\)
5: if \((t + i \cdot \text{size}) \leq \text{capacity}\) then
6: \(\text{itemsToObtain} \leftarrow \text{itemsToObtain} \cup i\)
7: \(t \leftarrow t + i \cdot \text{size}\)
8: end if
9: end while
10: return itemsToObtain
The P2R2 Distributed Algorithm dictates which items should be obtained in order to generate a replica. It does not specify how to get those items. In section 3.4, the procedure to actually obtain those items is explained. Inspecting line 33 from our implementation, it can be seen that issuing a request to download an item is subject to some condition. A naïve approach would be to issue an ItemDownloadRequestMessage if the item is not already contained in the peer, but this approach would lead to flooding the network with ItemDownloadRequestMessage messages. Algorithm 3 summarizes the steps we took in order to determine when peers issue item download requests.

The reason for the seemingly obvious check in line 4 is that, once an ItemDownloadRequestMessage has been sent, it is not known how long it will take for the request to be received. Therefore, at the time of issuing a new ItemDownloadRequestMessage, it is possible that a previously sent ItemDownloadRequestMessage has returned moments before and the peer issuing the request succeeded in obtaining a copy of the item.

Lines 1 and 2 refer to obtaining configuration values. It can be clearly seen that greater numbers of these parameters might lead to flooding the network with ItemDownloadRequestMessage messages, something that should
Algorithm 2 Distributed P2R2 Algorithm : receive

1:  itemsToObtain ← ∅
2:  if peer p receives request request for item i then
3:      p.itemsAwareOf ← p.itemsAwareOf ∪ i
4:  if request.ttl > 1 then
5:      request.ttl ← request.ttl − 1
6:  if p contains item i then
7:      if i is untouchable then
8:         request.satisfied ← true
9:         request.satisfiedBy ← empty_string
10:     else
11:        if request.isSatisfied = false then
12:           request.satisfied ← true
13:           request.satisfiedBy ← p.id
14:        else
15:           request.satisfiedBy ← min(p.id, request.satisfiedBy)
16:     end if
17:  end if
18:  end if
19:  if request.ttl > 1 then
20:     forward request to a random neighbor
21:  else
22:     send request backwards
23:  end if
24:  else
25:     if request.isSatisfied = false or request.satisfiedBy ≥ p.id then
26:        ri ← ri + 1
27:     end if
28:     if request.initiator ≠ p.id then
29:        send request backwards
30:     end if
31:     itemsToObtain ← knapsack(p.capacity, p.itemsAwareOf, r0, r1, ...ri)
32:     for all item k in itemsToObtain do
33:        if p.Items contains k then
34:           if p can issue a download request for k then
35:              send n ItemDownloadRequestMessage requests
36:           end if
37:        end if
38:     end for
39:  end if
40: end if
Algorithm 3 determining if an ItemDownloadRequestMessage can be sent

1: maxRequestsPerItem ← number of allowed requests per item
2: n ← number of neighbors to send a request to
3: if peer p should obtain a replica of item i then
4:   if p.items or p.untouchableItems do not contain i then
5:     currentRequests ← number of requests for item i contained in
6:       p.activeItemDownloadRequests
7:     if currentRequests < maxRequestsPerItem then
8:       r ← new ItemDownloadRequestMessage for item i
9:       send r to n different neighbors
10:   end if
11: end if

be avoided to give priority to other traffic.

3.1 The Item object

Everything around the P2R2 Distributed Algorithm revolves around making replicas of items. We use a simple class, Item that represents an item in the algorithm’s context. Each Item contains a system wide unique id and a size property.

3.2 Message structure

The P2R2 Distributed Algorithm relies on peers passing messages to their neighbors. There are several kinds of messages sent between peers and between the server components and peers. Connection objects can only process instances of classes that implement the PeerMessage interface. The PeerMessage interface extends the java.io.Serializable interface. This design decision implies that the state of any message that goes through a Connection object can be converted/recovered to/from a byte stream in order to simplify its transfer through network sockets. Figure 10 shows the most interesting messages that our implementation handles.

We offer a brief overview of selected messages.

- AdminMessage. Message sent from the NodeAdministratorService to the peers in order to either distribute a topology, distribute messages or request to take specific actions.
• **AbstractRequestMessage.** Messages extending this class inherit the logic of handling routed messages that have a time to live (ttl) counter.

• **NodeStatusUpdate.** This is the kind of messages sent from the peers to the server with information about the Node’s status.

• **ErrorMessage.** Since our implementation works simultaneously on several computers, it is extremely time consuming to read logs in order to find errors. These messages are built by remote peers whenever an error condition arises and sent to the server, where they can be inspected in one single file, or on the graphical interface server components.

### 3.3 Requests, special kind of messages

Several messages in the implementation are sent from the server components directly to the remote peers, or the other way around. However, there is a special kind of messages (i.e., the ones extending AbstractRequestMessage) that are not point-to-point messages, rather, they have a time to live counter (ttl) and keep track of the peers that have received the message. Whenever a peer receives one of these messages, it inspects the ttl in order to decide if the message is going forward (i.e., ttl > 1) or backwards (i.e., ttl <= 1) and act accordingly. Since P2R2RequestMessage and ItemDownloadRequestMessage extend AbstractRequestMessage, they don’t have the
need to implement this message routing logic.

A P2R2RequestMessage is a request that follows the P2R2 Distributed Algorithm, while ItemDownloadRequestMessage is devised for peers to actually download items from other peers. They both share the following fields:

- **id.** Each request contains a unique identifier. In order to guarantee uniqueness across the network and during the execution of the implementation, information such as peer *id*, request type along with a sequential number are used.

- **initiator.** The *id* of the peer issuing the request.

- **satisfied.** Whether the request has been satisfied or not. This field is interpreted in different ways for both kinds of requests, as following sections will explain.

- **satisfiedBy.** The *id* of the peer that satisfied the request.

- **visitedPeers.** Stack where the peers that have received the request are stored. In order to be able to send requests backwards, it is needed to keep track of the visited peers.

- **ttl.** Counter that each peer will decrement if the request is traveling *forward*. If a peer decrements this counter and its final value is 1, then the request will be sent backwards. Peers receiving a request with a *ttl* with value 1 will interpret the request as going backwards already.

### 3.4 Downloading items

Once the algorithm has determined that certain peer *p* must have a replica of item *i*, it is up to *p* to actually obtain that replica. The way we implement obtaining items is through sending messages with a certain time to live (*ttl*) counter. Peer *p* will send a message requesting to download item *i* to a random neighbor. Algorithm 4 presents the procedure followed by peers upon reception of such a message.

Note, in line 2, that the *isSatisfied* property is checked even before determining if the request is going backwards or forwards. This is because, no matter if the request is traveling backwards or forwards, the receiver of *request* will make its best effort to satisfy it.
Algorithm 4 item download : receive

1: if peer \( p \) receives an ItemDownloadRequest \( request \) for item \( i \) then
2: 
3: if request.isSatisfied = false then
4: 
5: if \( p \) contains item \( i \) then
6: 
7: end if
8: 
9: if request.isSatisfied = true then
10: 
11: send request backwards, along with item \( i \)
12: 
13: else
14: 
15: if request.ttl > 1 then
16: 
17: request.decrementTtl()
18: 
19: send request to a random neighbor
20: 
21: else
22: 
23: send request backwards
24: 
25: end if
26: 
27: end if
28: 
29: end if
30: 

If a peer receives a request that has not been satisfied yet and happens to possess the requested item (line 3), it will decrement the request’s \( ttl \) (line 5) in order to send it backwards as soon as possible, thus, avoiding further request forwards.

Whenever a request is satisfied, the item travels along with the request (line 9). This transfer of the item has been implemented as a waiting period, therefore, peers sending satisfied item download requests will have to actually \( wait \) for items to be transferred. The length of this “transfer” period is linearly proportional to the \( size \) of the item being transferred, as given in equation 1. \( K \) is a constant that is read directly from the configuration file.

\[
\text{transferPeriod}(item) = K \times (item.size) \tag{1}
\]

Transfer periods come in two flavors:

1. Waiting for some time by using \texttt{java.lang.Thread.sleep()}.

2. Waiting until an additional number of \texttt{P2R2RequestMessage} requests are sent by the peer transferring the item.

We have stated before that the set of properties of the computers used for our experiments is very heterogeneous. The experiments results should
not be interpreted as a function of time, rather, as a function of sent requests. The time needed to process a P2R2RequestMessage has an order of $O(n \log(n))$, where $n$ is the number of system wide items (more on this in section 3). So, for very large number of items in the system, the time taken to process a single P2R2RequestMessage increases significantly. Acquiring a copy of an item (i.e., transferring from one peer to another) is a very important step in data replication, since success rate depends on the existence of replicas across the peer network.

In order to be able to compare different experiment results (each experiment having different set of parameters), we use the success rate at a given number of sent queries. This means that, if we are simulating item transfer by using a timeout, (option 1), the results of the experiments will be affected by variables independent to the algorithm.

As an example, let us consider two experiments. The first experiment, $e_1$ uses a large number of systemItems (figure 11), while the second one ($e_2$) uses a very low number of systemItems (figure 12). Let us consider only one peer, $p$ and the time elapsed between events labeled $t_0$ and $t_1$. In $t_0$, for both $e_1$ and $e_2$, $p$ has sentQueries = 0 and satisfiedQueries = 0. As time goes on, $p$ downloads item0, item1 and item2 from other peers. Since both experiments are simulating item transfer by waiting some time, it takes the same amount of time to download these items. At exactly $t_1$, $p$ is able, in both experiments, to satisfy three requests (due to the downloaded items).

When the results from both experiments are compared, we see that the success rate at sentQueries = 10 is greater for $e_1$ than for $e_2$. A false interpretation of these results would lead to the conclusion that the more items we include in the system, the better the algorithm gets. Therefore, we introduced a time-independent option to simulate item transfers in order to factor request process time and avoid misleading results.

### 3.5 Node component

Besides containing the implementation of the P2R2 Distributed Algorithm, Node objects have the ability to send and receive messages to other remote peers concurrently. Each Node dedicates a special thread for messages coming from other remote Node objects and another thread for messages coming from the server components. As it has been previously discussed, one of the great advantages of using JXTA is that developers must not dedicate code to discover peers in the network. JXTA solves this problem by publishing XML
documents that can be discovered then used to establish connections between peers connected to the network. However, since we are interested in studying topologies with certain probabilities (e.g., edge probability), Node objects receive the topology from the server components. On a typical scenario, each Node maintains several reading and writing connections with other peers. Figure 13 shows how peers maintain communication.

All incoming messages from other peers are deposited into a queue and concurrently another thread is processing those messages. This design decision was done taking into account that having multiple threads processing incoming messages on each Node would not decrease process time, since each thread would have to be synchronized in order to access shared collections
Figure 13: Relationship between Node objects and remote peers

(e.g., data, statistics counters, and so on). Figure 14 shows how Node objects process messages.

Figure 14: Message processing relationships

We model Node objects following the specifications given in [6]. Each contains the following attributes:

- **id.** A unique identifier for each peer is generated based on the Internet Protocol Address and the port number used for communication.

- **untouchableItems.** Set of items that belong to the Node and that will not be modified by the algorithm.
• *items*. Set of items managed by the algorithm to place item replicas.

• *capacity*. Storage capacity of each peer dedicated to improve performance of the system. At any given time, for all peers, it is guaranteed that

$$\left(\sum_{i \in \text{items}} i.\text{size}\right) \leq \text{capacity}$$

• *$r_i$*. Each item known of, $i$, receives a counter stored under $r_i$. Please refer to section 3 for a detailed explanation of this set of counters.

• *itemsAwareOf*. Every time a *Node* receives a query for an item $i$, $i$ is included into this set.

• *itemQueryRate*$_i$. The probability to issue a request for an item $i$ is given by *itemQueryRate*$_i$. It is guaranteed that for each item $i$ in *untouchableItems* will have a query rate of 0. This is to reflect that peers won’t likely be requesting items that they already possess.

• *neighbors*. Set containing descriptors of other *Node* objects that can send/receive messages to/from this *Node*. This set is managed by the server components through the use of messages.

• *incomingPeerMessages*. A queue containing all incoming messages from other peers in the network.

• *incomingAdminMessages*. A queue containing all incoming messages from the server components. We keep separate queues because administrative messages have a greater priority to be processed.

• *activeItemDownloadRequests*. Whenever the P2R2 Distributed Algorithm mandates that a peer should obtain a replica of an item, each peer keeps track of each active *ItemDownloadRequestMessage*. An active request is defined as one that has been issued to a neighbor and has not been received back.

4 P2R2 within the Aeolus Testbed

The P2R2 Distributed Algorithm itself is efficient, compact and simple. In the proposed Java ™1.6 based implementation, few lines of code$^8$ were needed to implement it. The whole implementation, however, needs many more lines of support code$^9$. This is due to the fact that, besides implement-

$^8$93, to be exact

$^9$The implementation’s line code span is 6,813, divided among 78 classes
ing a distributed replication algorithm, we were interested in evaluating its performance under certain conditions through controlled experiments. Error handling routines, even though they are not an intrinsic part of the algorithm, are needed to guarantee, up to some extent, that our implementation can function under many circumstances.

4.1 Server components

Server components play a very important role in our implementation. They have in charge the critical task of collecting data and perform controlled experiments to evaluate the performance of the algorithm. Even though they manage remote Node objects, they don’t have a direct reference to them (i.e., resolved at compile time). In order to manage remote peers, they rely on the transfer of messages. Node objects were programmed to react accordingly for each message.

Server components are also regular peers connected to the Aeolus Testbed. They also benefit from the abstraction of sending and receiving messages by using instances of the Connection class. There are two server components that get executed in the Aeolus Testbed, namely, NodeAdministratorService and DataConsumerService.

Even though the server components are JXTA services, due to practical reasons, were designed taking into consideration that they will be hosted in the same Java™ 1.6 Virtual Machine (i.e., running under the same process). The responsibilities of each component are very different, therefore, communication between them is loosely coupled.

Since tasks such as remote data collecting, data processing and offering visual feedback on real-time (e.g., real-time status of the topology, real-time success rate chart, etc.) are not related to the P2R2 Distributed Algorithm, it has been decided to host them on a computer not managed directly under the Aeolus Testbed (i.e., our private computers). Through the use of scripts, we managed to hook up the server computer to the Aeolus Testbed network, thus, for some time our server computer was part of the Aeolus Testbed network (making them visible to the remote peers) and we could have full control of it.
4.1.1 The NodeAdministratorService

This is the component in charge of driving the experiments and collect the data coming from remote peers. The main tasks of this component are summarized as follows:

- **Topology management.** By sending messages to the peers in the network, `NodeAdministratorService` manages the connections of each registered peer. It also ensures that there is always at least one path between each pair of peers.

- **Collect data from peers.** `NodeAdministratorService` maintains a dedicated connection with each peer in the network, listening for update status messages coming from them.

- **Item management.** Peers constantly issue requests to the network asking for items. It is of special interest for our experiments to ensure that only requests for items that actually *do exist* in the system are issued. This means, that the `NodeAdministratorService` must generate, distribute and maintain a set of items by means of sending messages to peers to directly modify their `untouchableItems` set. Scenarios where peers issue requests for items that don’t exist in the network are outside the scope of our experiments.

- **Peer management.** `NodeAdministratorService` sends messages to remote peers requesting them to stop sending queries until further notice, to shutdown (i.e., terminate execution of the program) or to clear internal peer data.

In order to fulfill its tasks, `NodeAdministratorService` maintains and manages references of the following members:

- **incomingMessages.** All messages coming from remote peers are deposited in this queue for later processing.

- **systemItems.** This is the set of items that exist in the peer network. Each registered `Node` will get assigned a random number of unique items. The `NodeAdministratorService` ensures that, at any given time, all of the items in this collection are distributed among the registered peers.

- **topologyCreator.** An instance of the current used `TopologyCreator`. 
• Experiment configuration values. NodeAdministratorService loads data from configuration files in order to conduct experiments.

NodeAdministratorService itself does not compute statistics, as it is dedicated exclusively to manage nodes. Whenever a message containing peer data is received, it is forwarded to the DataConsumerService to be processed.

4.1.2 Topology management

In order to assist NodeAdministratorService with the task of maintaining a topology, the TopologyCreator was created. This component maintains a set of vertices (each peer is a vertex) and the edges existing between them (i.e., connections between peers). Figure 15 illustrates the design of the TopologyCreator.

![TopologyCreator Diagram]

Figure 15: Design of the TopologyCreator

So far, we were only interested in random topologies (i.e., topologies where each pair of vertices has an edge with a given probability edgeProbability). In order to use other topologies (e.g., ring topology, star topology, Internet style topology, etc.), it is only necessary to implement the TopologyCreator interface.

The ensureConnectivity method is of special interest for our experiments. In the case of a random topology, for small topologies or very low values of edgeProbability, it is possible to have at least one pair of vertices that don’t have any path connecting them. Figure 16 shows that there is no path for at least one pair of vertices (e.g., vertices b and f). The handling of topologies such as these are outside the scope of our implementation, therefore, in order to avoid these situations, the ensureConnectivity method

32
guarantees that each pair of vertices in the topology can be somehow connected by adding new edges, such as figure 17 shows it (the added edges are showed using a light gray dashed line).

Figure 16: A topology showing peer disconnections

Figure 17: A topology after invoking ensureConnectivity

4.1.3 Topology changes

Another important aspect that our implementation evaluates is how the algorithm performs in a highly churned network\textsuperscript{10}. As it has been stated before, Aeolus Testbed works based on allocations. Every time that an experiment is run, the allocation needed to be restarted. This means, that for every run of an experiment, we were limited to use the same computers that started the experiment. This restriction lead us to simulate churning as given in algorithm 5.

\textsuperscript{10}We define churning as the event where new peers join the network and peers already registered leave the network
Algorithm 5 churn $n$ peers

1: $\text{churnedPeers} \leftarrow \emptyset, \text{affectedItems} \leftarrow \emptyset, \text{affectedPeers} \leftarrow \emptyset$
2: broadcast a $\text{NodeActionRequestMessage}$ instructing nodes to stop sending requests
3: $\text{churnedPeers} \leftarrow \text{(select } n \text{ random peers from topology)}$
4: for all $p$ in $\text{churnedPeers}$ do
5: $\text{affectedPeers} \leftarrow \text{affectedPeers} \cup \text{topologyCreator.getNeighbors}(p)$
6: $\text{affectedItems} \leftarrow \text{affectedItems} \cup \text{getUntouchableItems}(p)$
7: send $p$ a message requesting to clear $\text{untouchableItems}$
8: end for
9: for all $p$ in $\text{churnedPeers}$ do
10: $\text{topology.vertices} \leftarrow \text{topology.vertices} \setminus p$
11: end for
12: while $\text{affectedItems} \neq \emptyset$ do
13: select random item $i$ in $\text{affectedItems}$, and remove it
14: select random peer $p$ in $\text{churnedPeers}$
15: send $p$ a message requesting to add item $i$ in $\text{untouchableItems}$
16: end while
17: for all $p$ in $\text{churnedPeers}$ do
18: $\text{topology.vertices} \leftarrow \text{topology.vertices} \cup p$
19: $\text{affectedPeers} \leftarrow \text{affectedPeers} \cup \text{topology.getNeighbors}(p)$
20: send $p$ a message requesting to clear $\text{items, itemsAwareOf}$ and $r_i$ and to reset $\text{itemQueryRate}_i$
21: end for
22: for all peer $p$ in $\text{affectedPeers}$ do
23: send a $\text{TopologyMessage}$ containing $\text{topology.getNeighbors}(p)$
24: end for
25: broadcast a $\text{NodeActionRequestMessage}$ instructing nodes to resume sending requests
When a Node is removed from the topology and reinstated once again, some of its collections are reset. These collections are the ones that are managed by the algorithm, thus, we emulate that a new Node joined the network. The fact that the items that the peers that we churn have been randomly redistributed and their itemQueryRate, has been reset emulates not only that a new peer has arrived to the network, but also that an already existent peer left the network.

4.1.4 Data collection from peers

The main metric we are interested on is the success rate, and it is defined as the ratio of satisfied queries\cite{6} to sent queries, globally. For this purpose, Node objects contain code to communicate to a centralized server. In a non-experimental application, these issues don’t exist, but we were not only interested on an efficient implementation, but also on evaluating its performance.

Typically, collecting data from peers distributed across a network, reduces to include code in each peer to communicate periodically (either every certain amount of occurrence of events, or every certain period of time) to a server, which in turn will collect and process the data. This sounds like a good enough approach. However, we must take into consideration that we are performing tests on a heterogeneous set of computers, this means, that each computer has different CPU speeds, RAM properties and so on. In other words, if peers send updates to the server every certain number of completed work-units\footnote{Server components receive updates about how much work peers have completed}, some peers will send updates more often than others, which would give misleading results when considering global success rate. On the other hand, if peers send updates every certain time, the server will get information from all peers at about the same time, however, the contents of those updates (e.g., number of satisfied and sent queries) will be affected by the computer characteristics. Figures 19 and 18 illustrate these concepts.

In order to perform a reliable evaluation for our experiments, we introduce the concept of a batchSize. Whenever Node objects send a number (batchSize) of queries, they will send an update\footnote{Updates include number of sent queries, satisfied queries, lost queries and the current collection of items possessed by the Node} to the server and wait until the server instructs them to carry on, as illustrated in Figure 20.
Using batches, we make sure that the received data corresponds to a global status, whereas in scenarios without using batches, statistics would be biased towards computers able to process more queries, thus it would be incorrect to speak of a global status.

4.1.5 The DataConsumerService component

The DataConsumerService uses an observer - observable approach. DataConsumerService "observes" for changes (i.e., messages coming from remote peers through the NodeAdministratorService) and is updated accordingly. Similarly, in order to decouple all other components interested in these
changes coming from the network, the DataConsumerService informs other registered components about these updates.

Communication between the DataConsumerService and the NodeAdministratorService occurs by means of the MessageCenter, which is a form of post office where components can post and read messages. As in post offices in real life, MessageCenter contains several mailboxes, each containing a unique ID that is used by the components. Whenever a component desires to post a message, it must provide, along with the message, the ID of the mailbox where this message will be sent. In a similar way, whenever a component needs a message, it must provide the ID of the mailbox where the message will be found. Delivery of messages is to a mailbox is a non-blocking operation, while reading a message from a mailbox might block if the mailbox is empty. This approach provides a highly decoupled communication between these vital components, as well as ease component testing.

In order to register a server component to receive updates, it must implement the java.io.Observer interface and be included in the configuration files. Upon start-up, the DataConsumerService, by using the Java™ 1.6 Reflection API, loads all components that will be observing changes. In this way, we decouple once again the logic of collecting messages from the logic of processing messages. Refer to figure 21 for a detailed representation of this process. The following sections describe the components created to assist in processing data collected by the DataConsumerService.
4.2 Graphical display

All graphical display components exist into a Console object. Components registered into the Console interpret and display data coming from the remote peers in a friendly, useful and graphical way. These components assist in the task of monitoring the performance of the system in real time, as well as detecting peers that might not be responding. The Console itself provides little functionality. Graphical Interface components can easily be plugged in by modifying the configuration file and, upon start-up the Console will instantiate each of these components and arrange them in the form of tabs, as figure 22 illustrates. The user can select which component to activate by clicking on a tab.

The Console behaves in a similar way to the DataConsumerService when it comes to inform registered components about an update. Updates to the Console come from the DataConsumerService. Upon reception of an update signal, the Console will inform its own observers (i.e., registered components).
4.2.1 Topology Overview Display

Through this component, it is possible to monitor the topology graph in real time. Each remote peer is displayed as a vertex and if two peers are connected, an edge is represented. Basic information about the peer is also available, such as its set of `untouchableItems` and `items`.

![Real time topology display](image)

Figure 23: Real time topology display

4.2.2 Real Time Charts

As a run of the experiments progress, it is of great interest to monitor its performance in real time. The plots displayed in this component can also be configured. Figure 25 shows a snapshot of the real time charts.

4.2.3 Remote Errors Display

Whenever testing new features or verifying bug fixes, sometimes remote peers stop responding due to the existence of exceptions not being handled. The main problem when handling errors is that the Aeolus Testbed offers access

---

13 Some experiments took as long as 8 hours to complete, therefore, a real-time feedback is important to assess the quality of the experiment.
Figure 24: Displaying peer information and highlighting its neighbors

to each peer’s log only through a web interface, this is, for each log, there is a http link; so, in order to read a log, it is needed to use a browser. Looking for an error under such conditions is extremely tedious, therefore, each peer sends a message to the server in case of the occurrence of an error, with enough information to locate the source of the error. Figure 26 shows the Error Display component in action.

4.2.4 Node Status Display

It is also important to know in detail the characteristics of a topology from the point of view of a single peer. A peer-centric view of the topology is displayed in this component. The displayed topology (see figure 27) takes the shape of several concentric semi-circles, where the peer of interest is shown in the middle, and the closer a peer is from the center, then the path needed to connect these peers is shorter. This component also offers basic search capabilities.
4.3 Report Generation

The system offers not only real time data visualization capabilities, it also generates report files that can be analyzed later. Similar to the Graphical Interface components, Report Generation components receive updates from the DataConsumerService (i.e., they are observers of DataConsumerService).

4.3.1 Data Reports

Statistics sent from the remote peers are received by the CsvReportGenerator component. It generates a standard comma separated value file (csv) that can easily be interpreted by plot generating or spreadsheet applications in order to process them. Each report file contains the names and values of the parameters set in the experiment, so it is possible to determine the characteristics of an experiment just by looking at the name of the file where the report is contained.
4.3.2 Topology Reports

Whenever the topology is created or modified (due to churning, as described in section 4.1.3), a file containing the topology is generated. The format used to generate these reports is GraphML[17]. This generated file can be later used to inspect the topology.

5 Evaluation

5.1 Parameters

All the way through our experiments, we used a topology of 50 computers with items of size 1. We used the following parameters as variables:

- **queryTtl.** Time to live counter of P2R2RequestMessage messages.
- **α.** Peers generate queries for item $i$ with a given $itemQueryRate_i$. The way we model the probability for a request for a certain to be issued is by using a Zipf distribution, which accepts a single parameter, $\alpha$.
- **Number of systemItems.** This is the size of the set of items that exist during the execution of an experiment.
- **edgeProbability.** Throughout the experiments, we use random topologies. The probability that two peers are connected is given by this parameter.
- **Number of churned peers.** Specifies the number of peers that will be churned, as explained in section 4.1.3.
5.2 Measures

For each experiment, we collect many data attributes, however, we are particularly interested in the success rate, which is defined as the ratio of total satisfied requests to total sent requests.
5.3 Methodology

We define a run as the execution of the implementation with certain parameters. For each experiment we performed, a parameter was selected and varied across runs, leaving the rest of the parameters fixed. We repeated the same experiment (i.e., using the same parameter values) three times and took the average of the three runs and with those results we generated a report file, in which each line contained one record. The presented plots were generated based on that report using the mean of the previous 32 records\textsuperscript{14}.

5.4 Experiments and Results

5.4.1 Variable number of items

We varied the number of systemItems using 100, 200, 400, 1,600, 3,200, 6,400, 12,800 and 25,600 items. The rest of the parameters were set as follows:

- $\text{ttl} = 15$
- $\alpha = 1.0$
- $\text{edgeProbability} = 0.2$
- $\text{capacity} = 12$

The results are shown in figure 28. We can immediately observe the success rate of a system is inversely proportional to the number of systemItems. Recalling from section 3, the idea of the algorithm is to generate replicas of items across the network.

Let us consider two similar systems, $S_1$ and $S_2$. $S_1$ contains more systemItems than $S_2$. The rest of the parameters have the same values. We assume, of course, that peers can store only a fraction of systemItems as replicas. Since the algorithm generates replicas of systemItems, we can intuitively deduce that in $S_2$ the algorithm will generate less replicas per item on average than in $S_1$. Due to the fact that peers, in both systems, can store only a fraction of the needed replicas, $S_2$ will have a greater success rate than $S_1$. Intuitively, the less items exist on a network, the more replicas per item on average can exist, therefore, the less items on a network, the greater the success rate can be.

\textsuperscript{14}This is known as a Simple Moving Average

44
Clearly, the success rate is directly proportional to the ration given in $2^{15}$: for a fixed peer capacity, the success rate of a system is inversely proportional to the number a system contains, because the number of queries that peers can satisfy is directly proportional to the number of replicas they can store.

$$f_{\text{items}} = \frac{\sum_{k \in \text{peers}} k.\text{capacity}}{\sum_{i \in \text{systemItems}} i.\text{size}}$$

(2)

Figure 28: Success Rate of a network for different number of items

15 More items stored as replicas means a greater number of satisfied queries
5.4.2 Variable storage capacity of each peer

The fixed parameters are as follows:

- $\text{ttl} = 15$
- $\alpha = 1.0$
- Number of $\text{systemItems} = 1000$
- $\text{edgeProbability} = 0.2$

We varied peer’s capacity across runs using the following values: 2, 4, 8, 12, 16, 20, 24 and 28 as figure 29 shows.

Similar to the experiment where the number of system items was varied, we are again facing a behavior that can be explained in terms of $f_{\text{items}}$ (refer to equation 2). We now keep the number of system items fixed and vary the capacity of each peer. For greater capacity values, $f_{\text{items}}$ will be greater, therefore, more requests will be satisfied. Figure 29 shows that the success rate is directly proportional to the capacity of the peers in the network.

5.4.3 Variable $\alpha$ parameter for the Zipf distribution

We varied $\alpha$ using 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4 and 1.5. The invariants for this experiment were

- $\text{ttl} = 15$
- Number of $\text{systemItems} = 1000$
- $\text{edgeProbability} = 0.2$
- Capacity = 12

From the results shown in figure 30, we can see that the success rate is directly proportional to $\alpha$. It has been stated that we have modeled each peer’s query rates for items as a Zipf distribution with parameter $\alpha$. In order to illustrate the effects of $\alpha$ on the query rate, refer to figure 31; the horizontal axis contains the rankings of items ($r$), while the vertical axis contains the probability of issuing a request ($p_{\text{request}}$) for items whose ranking is less or equal to $r$. So, for $\alpha = 1.3$, the first 4 items\(^{16}\) (0.4% of the total items) account for 50% percent of the queries, while for $\alpha = 0.7$, the first 138 items (13.8% of the total items) account for 50% of the queries. Clearly, for greater values of $\alpha$, fewer items will account for the majority of the queries\(^{17}\).

\(^{16}\)Items with a ranking of 4 or less
\(^{17}\)This is something discussed in detail by Zipf’s law
Figure 29: Success Rate of a network for different peer capacity

So, the number of items needed to generate the majority of queries is directly proportional to $\alpha$. Now, if a peer issues most of the queries for just a few items, other peers receiving these queries will tend to generate replicas for these few items (because their counters, $r_i$, will tend to have greater values for these items). Since all peers use the same $\alpha$, this phenomenon will occur across the whole network, meaning that for greater values of $\alpha$ is, fewer items will be needed to account for most of the queries. Having queries for just a few of items means that peers will be able to satisfy more queries,
because they will be able to store replicas for those items.

Another way to interpret the results is by thinking of query rates as a way in which peers express their favorite items (their top $- n$ items, so to say), assuming that peers will issue the majority of the queries for their favorite items. If the number of favorite items across peers is very low compared to the number of systemItems (i.e., greater values of $\alpha$), it means that, no matter how many items exist across the network, peers are interested in just a small fraction of them, therefore, queries will be issued for these favorite items. Since a peer will express interest for a few items, other peers will be able to accommodate replicas of those items easily.

5.4.4 Variable number of churned peers

We varied the number of churned peers, using 10, 20, 30, 40 and 50 as possible values, churning every 600,000 sent queries. The invariants are as follows:

- $ttl = 15$
- $\alpha = 1.0$
- number of systemItems = 1000
- $edgeProbability = 0.2$
- $capacity = 12$

The results are charted in figure 32. As discussed in section 4.1.3, whenever a peer is churned, we simulate that a peer leaves the network and a peer joins the network. Whenever a peer joins the network, it receives a set of untouchableItems and generates the itemQueryRate$_i$ set. We have already discussed that the replicas for items acquired by peers are directly proportional to the number of queries they have received for those items.

If a peer $p$ gets churned, its itemQueryRate$_i$ and untouchableItems sets will be modified, leading to a scenario where $p$ issues queries for a new set of items, thus, the probability for $p$ to get satisfied queries decreases. Other peers receiving queries from $p$ will try to adjust their items collection in order to generate replicas to be able to satisfy those queries, leading to a transient state of decreased success rate. Clearly, the more number of peers that get churned, the longer this transient state lasts, and the more severe the drop on the success rate it has, which is the behavior observed in figure 32.
5.4.5 Variable edge probability

As a variable for this experiment, we used *edgeProbability*, assigning values of 0.01, 0.02, 0.04, 0.08, 0.16 and 0.24. The invariants were:

- *ttl* = 15
- *α* = 1.0
- total number of *systemItems* = 1000
Figure 31: Accumulated probabilities of a Zipf distribution with several $\alpha$ values

- $capacity = 12$

The results are shown in figure 33. It can clearly be seen that the success rate of a system is directly proportional to the edge probability in which peers are connected.

The path that a query $q$ follows is the set of different peers that have received $q$, as given in definition 3

$$path_q = \{p_0, p_1, p_2, ... p_n\}$$  \hspace{1cm} (3)

If we consider two different issued queries for different items, $q_1$ and $q_2$, we can now define their intersection as the set given by

$$path_{q_1} \cap path_{q_2}$$

This intersection defines the peers that received both $q_1$ and $q_2$. We can further quantify how similar two different queries are as given in definition 4

$$pathSimilarity(path_{q_1}, path_{q_2}) = | path_{q_1} \cap path_{q_2} |$$  \hspace{1cm} (4)

For systems where peers have in average more neighbors, every time a peer $p$ issues a query $q$ for item $i$, the possible number of unique paths
Figure 32: Success Rate of a network for different number of *churned* peers, *churning* every 600,000 sent queries
that q can travel across is greater than for systems where peers have in average less neighbors; so, for any two different issued queries q1 and q2, \( \text{pathSimilarity}(\text{path}_{q1}, \text{path}_{q2}) \) is inversely proportional to the average number of neighbors peers have. Since the number of neighbors that a peer has is determined by \( \text{edgeProbability} \), for greater values of \( \text{edgeProbability} \), the possible paths that a query q can follow will be greater (because peers forward queries to a random neighbor); this means, as well, that more peers will be aware of the fact that peer p is querying for item i and it will be more probable that a replica of i will be placed among these peers and satisfy queries for item i.

Figure 33: Success Rate of a network for different edge probability

From figure 33 we observe that for values of \( \alpha \) greater or equal to 0.08 the success rate stops improving. Based on this observation, we can assume that there’s an upper boundary on the success rate if we vary \( \text{edgeProbability} \) and keep the other parameters fixed. The upper boundary is, in fact, the case where peers are connected using a mesh topology (i.e., \( \text{edgeProbability} = 1.0 \)). Figure 34 shows such a system (the rest of the parameters were set the same as in this experiment).
An intuitive way to explain this upper bound is that peers are limited to capacity restrictions when it comes to generate replicas. So, after certain value of edgeProbability, no matter how many neighbors peers have in average, the ability of satisfying queries comes from the items present in the untouchableItems set and the number of item replicas available, which is directly proportional to the capacity of the peers.

![Success Rate of a mesh network](image)

Figure 34: Success Rate of a mesh network

5.4.6 Variable query ttl

The results of this experiment can be seen in figure 35. We used, as a variable, the time to live, ttl, assigning it values of 3, 6, 9, 12, 15, 18 and 21. The rest of the parameters were fixed as follows:

- $\alpha = 1.0$
- total number of systemItems = 1000
- edgeProbability = 0.2
- capacity = 12

Figure 35: Success Rate of a network for different ttl

This experiment is related to the one described in section 5.4.5, Variable Edge Probability. ttl affects the path length that a query will have, therefore, for two queries q1 and q2, pathSimilarity(pathq1, pathq2) is inversely proportional to ttl. Thus, not only does edgePeer help peers to communicate their interest for items, but also ttl. A peer will be able to spread its interest for item i based on how many different peers are able to receive queries issued for i. Phrased another way, when edgeProbability is fixed, the number of
different peers that receive a query $q$ (namely, $path_q$, as given in definition 3) is directly proportional to $ttl$. This behavior is clearly reflected in figure 35.
6 Conclusions

We have presented an efficient implementation of the P2R2 Distributed Algorithm along with a system that assists in its methodical evaluation. During the experimentation phase, we were able to measure the effects of each parameter on the success rate of our P2R2 Distributed Algorithm implementation.

It is of utmost importance not only to identify the variables that can improve or impair the performance of a system, but also to gain an intuitive understanding of them, notions that we successfully acquired. After methodical and systematic experimentation we recognized the parameters that need to be adjusted or fine tuned in order to ensure a good performance of the algorithm under several conditions.

Keeping in mind the pitfalls that were identified in this document is a very important aspect when it comes to p2p systems and their experimental evaluation.

As a novel implementation of the P2R2 Distributed Algorithm, we have identified several areas of opportunity; not only to improve our implementation, but also for further research.

6.1 Future Work

It would be of special interest for our system to handle very large files, such as is the nature of video files. Naturally, peers should not be made to allocate a distributed copy of very large files, otherwise their capacities will be completely utilized very soon. An interesting approach would be to split large files in small pieces (chunks) and make replicas of those small pieces. Of course, deciding which chunks to make a replica of, and deciding the location to place its copy is not a simple task. Throughout our implementation, we assumed that queries were issued for a specific item; obviously, this approach would need to be modified in order to treat each chunk of a file as a separate item.

Due to the Aeolus Testbed’s restrictions, we were not able to test our implementation in topologies larger than 50 nodes. In p2p networks such as Gnutella, the number of connected peers is in the order of millions[19][21]. We would be very interested in observing how our implementation reacts to topologies whose diameter is longer than any random walk. A simple way to simulate this would be to go deeper on the granularity that the Aeolus
Testbed offers (i.e., having each virtual instance create sub-virtual instances to simulate more peers). However, communication in these sub-virtual instances cannot be compared to communication between virtual instances on the same computer or between different computers.

A different aspect that was not studied was that of topologies of dynamic size. In the early phases of development, we implemented a version in which peers would discover each other and refresh the list of neighbors every certain time. One of the strong points of JXTA is that, through the use of rendezvous peers, other peers have a great visibility, meaning that, for those early versions, we ended up with something very similar to a mesh topology. A simple approach would have been to code the peers in order to be connected to another peer based on the edgeProbability parameters. Nevertheless, for many other tasks we needed a central server to run the experiments and to manage nodes, plus, we wanted to have the chance to be able to test other topologies, and in order to be able to repeat these topologies, we needed a central server instructing peers which other peers they were allowed to communicate with. In order to test our implementation for topologies of dynamic size, we would have to do a serious refactoring of the code, or make the server components being able to handle these situations.

Another interesting direction of work would be to include a first order derivative in the $r_i$ counter. We have seen that one of the driving factors of replica generation is the counter $r_i$. Peers tend to have memory about which items used to have high query frequencies. Let us assume the existence of a system where all items have the same size, an item $i_1$ has a high query frequency, another item $i_2$ has a very low frequency and a peer $p$ has a replica of $i_1$. We subject this system to a churn and afterwards $i_1$ gets a low query frequency and $i_2$ gets a high query frequency. Peer $p$ will decrease the speed at which it increments the counter $r_{i_1}$ while the speed at which $r_{i_2}$ gets higher increments. So, if peer $p$ needs to generate a replica for $i_2$, its $r_{i_2}$ needs to be greater than any of the other items that exist in $p.items$, even though $i_2$ is getting an increased number of queries and $i_1$ is not getting too many new queries. Including the first order derivative of the $r_i$ counter could help to increase the speed in which a churned system to regains stability.

Throughout our experiments, we set the configuration values that were used as parameters in the algorithm. It would be really interesting to design

\footnote{That is to say, after churning, $i_1$ lost his popularity and $i_2$ gained popularity}

\footnote{Of course, this rate of increment is the first order derivative}
such a system that had some sort of awareness of its surroundings (neighboring peers, items contained in other peers, and so on) in order for peers to be able to auto-configure themselves. However, perhaps the most important item of future work would be to modify an open source implementation that relies on the Gnutella network[20] and include the P2R2 Distributed Algorithm. This would lead us to tap into large topologies[19] and study the Algorithm’s performance under real conditions.
List of Figures

1. Participants can connect computers to the Testbed .................................. 12
2. Each computer can run up to ten virtual instances ................................. 13
3. The Testbed showing several allocations .............................................. 14
4. Allocations are internally well connected and externally isolated .......... 14
5. Relationship between peers and Connection objects ............................ 16
6. Using the same connection to write and read ...................................... 17
7. Dedicating one connection to read ...................................................... 18
8. Dedicating one connection to write .................................................... 19
9. Exchange of a message between peers ................................................. 20
10. Messages handled in the implementation ............................................ 23
11. Network with a very large number of systemItems, depending solely on time to transfer an item ....................................................... 27
12. Network with a very low number of systemItems, depending solely on time to transfer an item ....................................................... 27
13. Relationship between Node objects and remote peers ....................... 28
14. Message processing relationships ....................................................... 28
15. Design of the TopologyCreator ............................................................ 32
16. A topology showing peer disconnections ............................................ 33
17. A topology after invoking ensureConnectivity ................................... 33
18. Sending updates every certain number of work units ............................ 36
19. Sending updates every certain amount of time .................................... 36
20. Sending batches ................................................................................... 37
21. Forwarding collected remote data to other components ..................... 38
22. Loading graphical display components into the console ..................... 38
23. Real time topology display .................................................................. 39
24. Displaying peer information and highlighting its neighbors ............... 40
25. Real time chart generation ................................................................... 41
26. Remote error display ........................................................................... 42
27. Peer-centric topology as shown by the Node Status Display ................ 43
28. Success Rate of a network for different number of items ................. 45
29. Success Rate of a network for different peer capacity ....................... 47
30. Success Rate of a network for different α ........................................... 49
31. Accumulated probabilities of a Zipf distribution with several α values ....................................................... 50
32. Success Rate of a network for different number of churned peers, churning every 600,000 sent queries ............................................. 51
33. Success Rate of a network for different edge probability ................... 52
34. Success Rate of a mesh network ......................................................... 53
35. Success Rate of a network for different ttl ........................................ 54

59
### List of Algorithms

1. knapsack\(\left(\text{capacity, itemsAwareOf}, r_0, r_1, \ldots, r_i\right)\) .......................... 19
2. Distributed P2R2 Algorithm: receive ........................................... 21
3. determining if an ItemDownloadRequestMessage can be sent . 22
4. item download: receive ................................................................. 25
5. churn \(n\) peers ................................................................. 34
References

[1] Symposium of Principles of Distributed Computing, PODC
   http://www.podc.org/

[2] Towards a complete grid filesystem functionality
   Soha Maad, Brian Coghlan, Geoff Quigley, John Ryan, Eamonn Kenny, David
   O’Callaghan

   http://www.majestic12.co.uk/

   http://grub.org

   http://yacy.net/

   Mauro Sozio, Thomas Neumann, Gerhard Weikum

[7] The Apache Ant Project
   http://ant.apache.org/

[8] The Prefuse Visualization Toolkit
   http://prefuse.org/

[9] JXTA
   https://jxta.dev.java.net/

[10] Apache log4j
    http://logging.apache.org/log4j/1.2/index.html

    http://jchart2d.sourceforge.net/

[12] Aeolus Testbed
    http://aeolus.cs.upb.de/

[13] Aeolos Project
    http://aeolus.ceid.upatras.gr/

[14] Aeolus Installation Software
    http://aeolus.cs.upb.de/testbed-2.1.exe
    http://aeolus.cs.upb.de/protected/downloads/testbed.zip

    Sun Microsystems, 2002-2007
[16] JXTA-JXSE 2.4.1
http://download.java.net/jxta/jxta-jxse/2.4.1/

[17] The GraphML File Format
http://graphml.graphdrawing.org/

[18] The Java Interactive Profiler
http://jiprof.sourceforge.net/

[19] Study: BitTorrent sees big growth, LimeWire still #1 P2P app
http://arstechnica.com/old/content/2008/04/study-bittorrent-sees-big-growth-limewire-still-1-p2p-app.ars

[20] Limewire
http://www.limewire.com

Amir H. Rasti, Daniel Stutzbach, Reza Rejaie

[22] Gnutella for Users
http://rakjar.de/gnufu/index.php/Main_Page

[23] The Gnutella Protocol Specification v0.4
http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[24] Gnutella2 Developer Network
http://g2.trillinux.org/index.php?title=Main_Page

http://cvs.berlios.de/cgi-bin/viewcvs.cgi/gift-fasttrack/giFT-FastTrack/PROTOCOL?view=markup&content-type=text%2Fvnd.viewcvs-markup&revision=HEAD

[26] Replication Strategies in Unstructured Peer-to-Peer Networks
Edith Cohen, Scott Shenker

[27] Search and Replication in Unstructured Peer-to-Peer Networks
Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker

[28] Creating and maintaining replicas in unstructured peer-to-peer systems
Elias Leontiadis, Vassilios V. Dimakopoulos, Evaggelia Pitoura

[29] Efficient Lookup on Unstructured Topologies
Ruggero Morselli, Bobby Bhattacharjee, Aravind Srinivasan, Michael A. Marsh

[30] ANDREW: A Distributed Personal Computer Environment
[31] OpenAFS  
http://www.openafs.org/

Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey  
M. Voelker

[33] Efficient Replica Maintenance for Distributed Storage Systems  
Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, Robert Morris

[34] Designing a DHT for low latency and high throughput  
Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, Robert Morris

Shanyu Zhao, Daniel Stutzbach, Reza Rejaie

[36] Copyright and Peer-To-Peer Music File Sharing: The Napster Case and the  
Argument Against Legislative Reform  

[37] The Freenet Project  
http://freenetproject.org

[38] The DOE Science Grid  
http://www.doesciencegrid.org/

[39] SETI@home project  
http://setiathome.berkeley.edu/