Distributed File Replication System based on FreePastry DHT

Franjo Plavec and Tomasz Czajkowski
Edward S. Rogers Department of Electrical and Computer Engineering
University of Toronto, 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4
{plavec, czajkow}@eecg.toronto.edu

Abstract

Peer-to-peer networks provide a framework appropriate for building various distributed applications. One of such applications is a distributed file replication system. File replication ensures data availability in face of system component failures. In this paper, a distributed file replication system based on a distributed hash table, implemented as part of the course project, is described. The system supports basic operations provided by the traditional file systems. The protocols used for data replication and consistency are analyzed, and the implementation details are discussed. The methodology used to test the system performance is described, and the measurement results are presented and discussed.

1. Introduction

Peer-to-peer networks are becoming increasingly popular for applications such as file-sharing, content distribution and distributed storage. The advantage of using peer-to-peer systems stems from the inherent capabilities of these nodes that are otherwise unused most of the time. For instance, previous research showed that over half the disk space on desktop workstations in Microsoft was unused [1]. The disadvantage of peer-to-peer systems is that most of the nodes in the system are transient, and may be available only for short period of time. Therefore, such systems have to provide means for dealing with high churn (network nodes joining and leaving constantly) in the system. For the distributed storage systems, this means that data replicas have to be maintained on multiple nodes. If a set of nodes that stores replicas of any given piece of data are diverse enough in terms of geographic location, network type and legal authorities, the system can provide availability guarantees with high probability. In fact, peer-to-peer systems provide better availability guarantees on a global scale.

One of the ways to implement a peer-to-peer network is by using the functionality of a distributed hash table (DHT). A distributed hash table performs the functionality similar to that of a hash table. Given a key, DHT provides the location (a network node) where the key-value pair is stored. This primitive can be used as a base for various distributed applications.

In this work, we use the Pastry DHT to implement a distributed file replication system. Each node in our system acts as a client, issuing data requests, and as a storage node, storing a portion of the system data. The nodes can join and leave the system at any given time, without warning. To maintain data availability, files are replicated on multiple nodes. The system adapts to the changes in the system by migrating data among nodes and generating new file replicas to keep the number of replicas in the system constant. The system provides strong guarantees on the data availability. Unless all the nodes storing a file fail in a short period of time, the system is guaranteed to make the file available on at least one of the nodes.

All the nodes holding the replica of a file can respond to the user requests. This means that simultaneous updates to a file may occur on two different nodes. Possible conflicts that occur due to this are handled by the consistency protocol. In our system, we use a simple versioning consistency protocol, which ensures that, if the system is left undisturbed for some period of time, all replicas of a file in the system will eventually be consistent.

Our system was deployed on a real computer network for the purpose of testing. We measured the system performance in terms of response time and network traffic, as well as the resilience to node failures.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work. In section 3 we describe Pastry, a DHT our system is built upon. Section 4 gives an overview of our system design and implementation details. Section 5 presents our experimental results and discusses the results. We conclude in Section 6, and give an overview of future work in Section 7.

2. Related Work

There has been an extensive amount of research conducted in the area of distributed hash tables and data replication. In this section we give an overview of several DHT systems, replication systems and consistency schemes.

Aside from Pastry, there are several DHT systems available. Tapestry [2] routes messages incrementally using prefix matching algorithm. Tapestry supports replication, but does not replicate the objects themselves, but rather pointers to the node that stores objects, thus achieving faster response times. Data itself can also be replicated, in which case pointers to the data contain pointers to all copies of the data.

Chord protocol [3] provides a key lookup service comparable to the message delivery in Pastry. Given a key, the Chord system will find the node responsible for the key.

Node identifiers and keys are generated using SHA-1 hash function. A key resides on the node whose identifier is equal to or successor of the key in the identifier space. If the identifier space has 2^m elements, the routing table on each node consists of m entries. The routing table of a node with id n is organized such that the i^{th} entry contains the address of the node that is responsible for the key $(n+2^{(i-1)}) \bmod 2^m$. Therefore, the node contains more information on the nodes that are close to it in the identifier space than those that are far. The messages are routed based on the numerical difference to the destination address.

Chord can support data replication by storing a list of r nearest successors on each node. Due to the routing scheme, this method of replication does not achieve load balancing, only data redundancy. [3] suggests that the Chord system is expected to be used as a lookup service in the "rendezvous" process, rather than for file transfer.

[4] presents Content Addressable Network (CAN), which is a form of a distributed hash table. The peer nodes in CAN are placed in a virtual, d-dimensional Cartesian coordinate space. Each node is responsible for a portion of the space called a zone. Keys are also mapped to the same d-dimensional space and stored on a node responsible for the zone the key falls into. Each node stores information about O(d) other nodes in its routing table. Routing is implemented by approximately following the straight line in the coordinate space, on average requiring $O(d*n^{(1/d)})$ hops in a network with n nodes. The routing table therefore does not grow with the network size, but the number of routing hops grows with the number of nodes faster than log n.

PAST [5] is a replicated storage system built on Pastry. The files in PAST are immutable, and although the storage taken by a file can be reclaimed when the file is no longer needed, traditional semantics of delete operation is not fully supported. PAST relies on the locality characteristics of Pastry. The files in the system can be encrypted. Files are assigned unique IDs generated by hashing the file name, the owner's public key, and a random number using the SHA-1 hash function. No file search facilities are provided. A file can only be accessed if the correct file ID is presented to the system. Each file can be replicated on multiple nodes, depending on the expected popularity of the file. In addition, the system caches popular files on additional nodes. Unlike the replicas of a file, the cached copies may be discarded at any time if the node needs the storage space to store other files. Cache replacement policy considers the popularity of the file, its size, and the capacity of the node caching the file.

To accommodate the diversity in storage and processing capabilities of the nodes, the PAST system allows replicas of a file to be stored on nodes other than the r nodes that are closest to the file ID. This process is called replica diversion. In case of node joins/departures, portion of the file is migrated gradually to the newly arrived node, similarly to our system. However, the newly arrived node in the meantime stores the pointers to the nodes that actually contain the data, which is not the case in our system. PAST is not designed to be used as a general-purpose file system, but rather as an archival storage and content distribution system [5].

Active Directory [6] is a directory service built into the Windows 2000 operating system. Active Directory storage is organized as a hierarchical database, and is primarily used for storing information about the network resources in Windows 2000 network, although it can be used to store custom data [7]. The data is replicated on multiple computers in the network. Replication framework takes the topology of the network into account when determining the frequency of updates among replicas. The network administrator defines Active Directory sites, which correspond to the well connected computers in the network. Replication is then performed more frequently inside the site than among the sites.

The consistency model used in Active Directory is called multi-master loose consistency with convergence [8]. Multi-master refers to the fact that the object may be updated on any of the nodes that hold a replica of the object. Loose consistency means that the replicas are not guaranteed to be mutually consistent at any given time. However, if an object is not changed on any of the nodes for sufficient amount of time, the system converges to a stable state, where all the replicas are consistent (converge). We use this consistency scheme in our work.

The consistency protocol uses version numbers, which are in Active Directory referred to as Originating Write properties [6]. Each time an object is modified by the client application, the Originating Write property is incremented. By storing the Originating Write property and a GUID (Globally Unique Identifier), the system can resolve all conflicts in record versions. The Originating Write Property is not updated when the file is replicated, or version reconciliation occurs. In our system, we use the node ID as a GUID, as described in Section 4.2.3.

Alternative consistency scheme using version vectors is described in [9]. In this scheme, instead of storing only one version number, as in Active Directory, each file has associated version vector, with one component per node that holds a replica. Every time a node updates its replica, it increments the version number in the vector corresponding to that node. [9] shows this information is sufficient to detect conflicts. When conflicts are detected, it is up to the application to decide on the reconciliation mechanism, depending on the semantics of the operations that were performed on the files. This is in contrast to the Active Directory approach, where the reconciliation mechanism is encoded into the system. During the reconciliation, a new vector number is produced by taking the maximum version number for each component, and finally incrementing the version number for the node that performs reconciliation. Similarly to our system, [9] treats file deletions as file

updates by increasing the appropriate component of the version vector, and setting the file size to zero.

3. Pastry

We build our replication system on top of FreePastry, which is an implementation of Pastry protocol. Henceforth we will refer to this implementation as Pastry for clarity. In this section we discuss the basic operation of Pastry [10][11].

Pastry is an implementation of a DHT. This implementation consists of nodes connected using a peer-to-peer network. When a node joins the system it is assigned a unique ID. Each node ID is a 160-bit value. In this 160-bit ID space we define the distance between nodes A and B to be $min(A\text{-}B \bmod 2^{160}, B\text{-}A \bmod 2^{160})$. This makes the address space circular. Thus, the distance between nodes is a minimum distance along a circle from one node to another.

In addition to providing node IDs to nodes in the system, Pastry facilitates communication between nodes. The communication strategy is demonstrated in Figure 1. To send a message from a source node Q with node ID 2012 to an address 0124, Pastry first sends the message to a node that has a matching prefix of one digit. The recipient checks if the message is addressed to him, ie. the recipient has node ID closest to the destination address. If not, the recipient forwards the message to a node that matches the destination prefix with one more digit. By using this forwarding scheme the message arrives at its destination in log (N) hops. As shown in the figure, a node sending message to address 0124, first sends it to any node that has first part of the address 0. Then the message is forwarded to a node that has prefix 01, then 012. Since the last node ID digit cannot be matched by any node, the node 0122 processes the message as the node 0122 has the closest node ID.

To route messages between efficiently, each node in Pastry stores a routing table, leaf set and the neighbourhood set. The routing table contains as many rows as there are digits in a node ID, and the number of rows is one less than the number of distinct digits in a given base (we use base 16). The n^{th} row of the routing table lists nodes that have matching prefix of length n with the current node ID.

The leaf set contains addresses of nodes whose IDs are close to the current node's ID in the identifier space and is used in routing. Finally, the neighbourhood set contains addresses of nodes that are close to this node in terms of proximity metric. Instead of using this table for routing, this table is used for routing table updates when the state of the system changes.

Now that we have described the DHT system we base our design on, we proceed to describe the design of our file replication system.

4. Design

A distributed file replication system provides the user with the ability to store and retrieve files reliably, even when some of the machines storing the data fail. The user interacts with the file replication system via a user interface. The user interface communicates user requests to the system and returns responses to user queries.

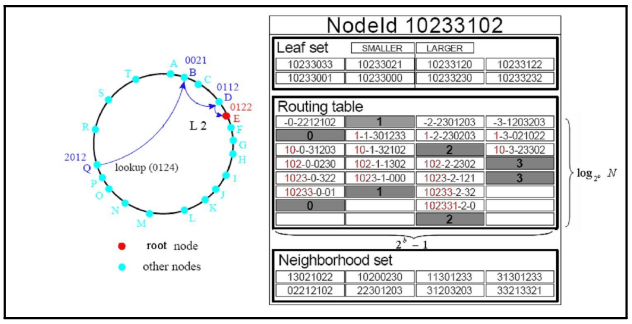

Figure 1: FreePastry Data Structures

In this section we describe the implementation of the distributed file replication system and give a general system overview. Next, we describe main components and their operation.

4.1 System Overview

Our distributed file replication system consists of a set of replication nodes that are connected by a peer-to-peer network. The network nodes extend the functionality of Pastry DHT nodes.

The set of replication nodes is responsible for storing and maintaining all files for the system as well as a specified number of replicas for each file. Each file in the system is stored on a subset of replication nodes. If one of the replication nodes fails, the set of replication nodes must adjust the data stored on all replication nodes in such a way as to retain specific number of replicas for each file.

Each replication node extends the functionality of a DHT node. In addition to the basic DHT node functionality, a replication node manages file storage and issues replication messages to other replication nodes as necessary.

In the following sections we describe the operation of each replication node, the replication strategy, messages exchanged between replication nodes, how the system behaves when nodes join and leave, and the periodic update mechanism.

4.2 Replication Nodes

Replication nodes provide two functions in our system. First, they store and manage files. Second, each node is responsible for handling message passing through it and responding to nodes joining and leaving the system. To do this, we divide the replication node into three modules, as shown in Figure 2. These modules are the message handler, the replication handler and the consistency and storage handler. We describe these modules in the following subsections.

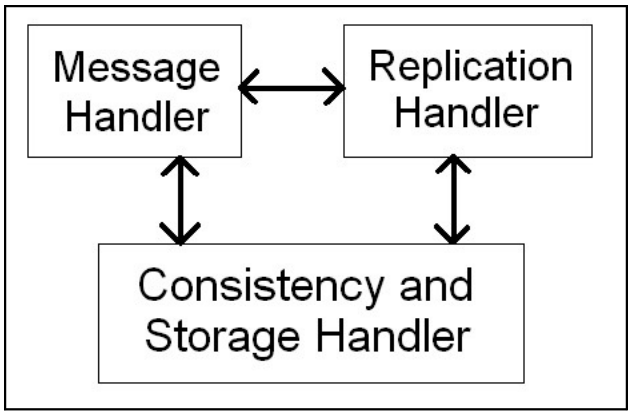

Figure 2: Replication node components

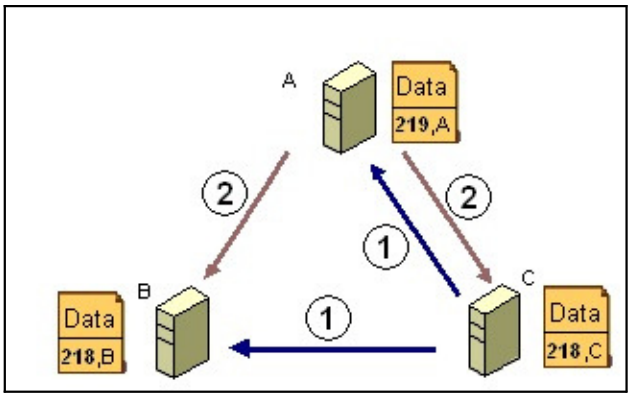

Figure 3: Ensuring consistency using <version, nodeID> tuple

4.2.1 Message Handler

The message handler is responsible for processing messages that arrive at the replication node. These messages may either be intended for this node, or simply on their way to another node.

When a message passes through a node, the message handler checks its destination address. If the address of the message and the address of this node match then the message is handled by this node. Otherwise, a node may change a message and forward it to the next node, until the message arrives at its destination.

Alternatively, the message handler may determine that the message intended for another replication node can be handled by this node. In such a case the message handler will attempt to respond to the message, by deleting, reading or writing to a specific file. To create a file, the create message must arrive at a node that has a node ID that is the closest match to the **file ID**, which is created by hashing the file's name using SHA-1 hash function.

Once the operation is complete the message handler will send a response back to the sender and inform the replication handler if it is necessary to update other copies of the same file on other replication nodes.

4.2.2 Replication Handler

The replication handler is a module responsible for sending replication requests to other nodes. This module is activated by either the message handler, a notification from the DHT that a node has joined or left the system, or a timer.

The message handler activates the replication handler in a case when a message to create, write, or delete a file arrives at this replication node. When the message handler activates the replication handler, the replication handler will determine which nodes contain, or should contain, the data regarding the modified file and send messages to update other replication nodes.

The replication handler may also be activated by the DHT, when the DHT detects that a node has joined or left the system. In such a case it is the responsibility of the replication handler to either move files from one replica to another, or to create replicas of the file that has insufficient number of copies. We describe the join and leave procedures in more detail in section 4.6.

Finally, this module may be triggered by a timer. A timer has been set up to allow the system to periodically update all files in the system, thus providing loose consistency. We describe the mechanism for periodic updates in more detail in section 4.4.

4.2.3 Consistency and Storage Handler

The consistency and storage handler is responsible for storing files for a particular replication node and ensuring that local files are only replaced with newer versions of the same file. The key to this effort is a file versioning.

Versioning [8] is a method to keep track of a status of a file, by storing its version number. We use an approach similar to [8] in this work.

In our system file versioning is denoted by a <version, nodeID> tuple. The file version number determines the number of times the contents of the file were modified. A file with higher version number is considered to be the most up-to-date. The create operation sets the version number to 1, while each write operation increments the version number. The delete operation is treated as a write operation that clears the contents of the file.

The nodeID part of the tuple specifies the node ID of the node that performed the last modification to the file. In a case that a write or delete message arrives with the file version number the same as the local file copy, we compare nodeIDs for the message and the local file. If the IDs are the same, then the local file copy is up to date. This means that the update message must have been generated as a result of updating the file on this node, in which case the local file copy is up to date. If the message that arrived has a higher nodeID then we update the local file copy.

Consider the example in Figure 3. Nodes A and C are trying to replicate their file named *Data*. First C sends its file with version number 218 and ID equal to C. Node A ignores replication request because it has a newer version of the file, but node B accepts it because of higher ID, so B

Table 1: List of messages in the distributed file replication system

Message Type	Function	Communication Type
Create	Create a file	Indirect
Write	Write data to a file	Indirect
Read	Read data from a file	Indirect
Delete	Delete a file	Indirect
Read Reply	Respond to a read file request	Direct
Update	Update file on a particular replica	Direct
Acknowledge	Respond to write, delete and create requests	Direct
Create Replica	Issue replica creation for a file	Direct
Delete Replica	Delete a replica of a file	Direct
Acknowledge Replica Creation	Respond to a replica creation request	Direct
Acknowledge Replica Deletion	Respond to replica deletion request	Direct
Acknowledge Replica Update	Respond to replica update request	Direct

stores file version <218,C>. Next, node A sends its file, which replaces files on nodes B and C. In the final state of the system, the data file version <219,A> is on all nodes.

4.3 Replication Strategy

To make the data available in the system, despite nodes joining and leaving the system, we provide a mechanism to maintain sufficient number of replicas for each file. In the case of a node that joins, we move some files from other replicas to it. On the other hand, when a node leaves, some files have insufficient number of replicas in the system. To remedy this we considered two replication strategies: **0-root** replication and **n-root** replication. We now describe both strategies.

In the 0-root replication strategy the system only allows the node with node ID closest to file ID for a particular file to send replication messages to the newly arrived node. We refer to such node as 0-root node. This limits the network traffic and ensures that only one replication request per file is sent to any node. However, if several nodes join and leave the system in a short period of time, then it is possible for files to become unretrievable.

For example, consider three nodes (A, B and C) in a system with replication factor 3. Let each node contain files (a, b, c) and node A be 0-root for file *a.txt*, node B be 0-root for file *b.txt*, and node C be 0-root for file *c.txt*. Now consider that nodes B and C fail and node D joins the system at roughly the same time. Because of this node A becomes the 0-root for file *b.txt* and node D becomes the 0-root for file *c.txt*. However, node D does not have file *c.txt*. Thus, file *c.txt* will be unreachable on node D and node A will have no incentive to communicate this information to node D. To permanently lose file *c.txt* all it takes if for node A to fail.

To remedy this problem we employ n-root replication. With this strategy each node that has a file that does not have enough replicas will send out messages to nodes that should have a replica of the file. This causes more network traffic, but a file will be replicated even if all but one replicas fail and unspecified number of nodes joins the system.

The drawback of this strategy is that it can potentially require high bandwidth to transfer large files as many times as there are replicas in the system. We leave the problem of reducing the traffic associated with this strategy as a future work.

4.4 Inter-node messages

In previous sections we explained the operation of each replication node and how each replication node behaves in the context of the file replication system. We now discuss how the nodes communicate with one another.

To communicate information between nodes, replication nodes send messages to one another. The different message types are summarized in Table 1. Each message in our replication system is associated with a communication type. The communication type defines if a message is sent *indirectly*, using the standard Pastry protocol, or arrive at the destination *directly*.

The indirect communication type is used when requesting access to a file. Due to replication a message may pass through a node that holds a replica of a requested file allowing the intermediate node to respond to the message, which reduces the response time.

On the other hand, when the system needs to synchronize the data, it requires direct responses from specific nodes. For the correct functioning of the system, one node may not respond in place of another in such a case. Thus, we implement direct communication for the purposes of data replication. Direct messages are only sent by the system itself, whereas the indirect messages initiated by the user.

We now describe user and system messages in more detail. We then describe how the system behaves when a new replication node joins a system or an existing node fails and leaves the network.

4.4.1 User Messages

In our system the user can use four types of messages: *create*, *write*, *read* and *delete*. The create request asks the system to create a file with specified name within the system. The created file will have length 0 and will be replicated as soon as the system is able to do so. To store data into a file, the user must send a write message. The write message associated with a file contains the data to be written to that file. The user may also remove a file from the system. To do this the user sends a delete message for a particular file. The file will be deleted from the system if it exists. Otherwise, the system will respond with an error message.

In response to a create, write, or a delete message, the system returns an *acknowledge* message back to the node that first received the user request. This message specifies if the operation was successful or not.

Data stored in the system can be retrieved by issuing a read request. When a read request message is received by the system, it will be forwarded to a node that contains the specified file. Upon successful retrieval of the file, the system will send back a *read reply* message containing the data stored in the specified file.

Despite the presence of a mechanism to send and receive messages, it is possible in any network application for messages to be lost or dropped. This happens mostly due to network congestion. For that reason it is necessary to have a mechanism to resend the message periodically to allow it to reach its destination. To do this we use a **resend queue**.

The resend queue stores all user messages sent by a replication node. Before sending out a user message, the message is added to the resend queue. The queue is periodically scanned by the replication node and if a reply is not received within 30 seconds, the message is resent under the assumption that it was dropped in transit.

Each message stored in the resend queue is assigned a time-to-live (TTL) which specifies the number of times the message can be resent. This prevents messages from being sent to dead nodes or to nodes that are heavily congested.

After one of user messages is entered and processed by the replication system, there are a number of system messages that are sent between replication nodes. These message facilitate replication to ensure that files are not lost in the system and that the correct number of replicas of each file exists in the system. These messages are described in the following section.

4.4.3 System Messages

System messages comprise the bulk of messages sent within our system. These messages are sent any time a file is modified to ensure consistency. System messages are the means by which our system eventually converges to a state where all replicas of a given file are up to date. These messages are: *create replica*, *update*, *delete replica*, *acknowledge replica creation*, *acknowledge replica update*, and *acknowledge replica deletion*. By using a pair of these messages to create, update and delete a file from the system, we ensure loose consistency within the system.

To create, update or delete a file a replication node first determines which replication node to communicate with. This is determined by the replicasSet() function provided with Pastry[11]. The inputs to this function are the fileID and the replication factor. The function returns a set **S** of nodes that should hold a replica of a given file.

To create or update a file replica a create/update replica message is sent to all nodes in set **S**. A node that receives a create/update replica message will first check if the local file is newer than the one specified by the message. This is determined by versioning, as described in Section 4.2.3. If the local copy of a file is old, the file will be updated and a message will be sent back to acknowledge replica creation/update. In the case when the local file is newer, the system will still send an acknowledgment message, but in addition it will send out its own version of the file to all replicas holding this file.

To delete a file we follow a similar procedure. However, to delete a file it is insufficient to simply send a message to all replication nodes storing this file. For example, consider nodes A, B, and C that all contain file 100.txt. Let a delete file request arrive at node C. This node will then proceed to delete its own copy of the file and issue delete replica requests to nodes A and B. Suppose that node A never receives this message because of network failure and some time later attempts to update node B with its own copy of the file. Node B will view this as a request for file replication and will communicate this information to node C, effectively recreating the file.

To prevent this we introduce **tombstones**. A tombstone informs a node that held a file that the file has been deleted. In addition to the knowledge that the file has been deleted, the tombstone holds a list of all nodes where the file is still stored. While a node is aware of another node that still stores a file, the tombstone will be kept. This will force the system to periodically attempt to delete the replica of a file on nodes that have not positively acknowledged replica deletion.

4.4.4 Node Join and Leave

Replication nodes may join or leave the system at any time. In either case, the system must adjust by either moving files from one node to another or by creating more copies for files that were stored on a node that left.

When a node joins the system it first contacts the bootstrap node of the system. The bootstrap node is a node that is known to exist in the system. The newly arrived node will then negotiate with the bootstrap node to get a node ID and find its place in the system. Once this is done, the new node will build its routing table, leaf set and a neighbourhood set by communicating with the bootstrap node. Once these three data structures are created, the new

node will announce its arrival to all nodes in the system by sending each of them a join message.

When a replication node receives this message, it checks all its files to determine if any of the files should be replicated on the new node. This means that all nodes that have a copy of the file that is to be moved will send the file to the new node. While this creates a lot of traffic, it is necessary to ensure that at least one node that stores a copy of the file communicates with the newly joined node.

When a node leaves the system, it takes some time for the system to detect this. To detect if a node has left the system, each replication node sends heartbeat messages to its neighbours in the ID space. If it does not receive a response from one of these nodes for some time, it will assume that the node has left the system. When it detects this, a message will be sent to other nodes to inform them that a node has left the system.

Each replication node that receives this message must determine if any of the files it stores has enough replicas. For all files that do not have enough replicas in the system, a replication node issues a replication request to make more copies of the same file on a node that does not currently hold the file.

While in the case when all messages arrive safely at their destinations the system works well, in practice however heavy network traffic or failed network links may cause messages to never arrive at their destinations. To ensure eventual convergence we perform periodic updates, which we describe in the next section.

4.5 Periodic Updates

It is possible for system messages to be dropped in transit while in the network. Failing to deliver these messages, the system would never converge, which presents a problem for a file replication system. To avoid this problem we periodically force the system to behave as though a new node has joined the system.

For each file we store a record of replicas that have responded to a create, update or delete message. Each replication node that has not responded to a system message is treated as if it were new to the system, thus forcing another replication node to send it a file update.

5. Experimental Results

In previous section we described the distributed file replication system based on Pastry DHT. We have shown the different types of operations available in our system and described the implementation details, justifying design decisions we made. In this section we show how our system performs when deployed on a real network platform.

In the following subsections we describe our experimental setup, testing methodology and the results we measured. We then discuss the measurements we obtained.

5.1 Experimental Setup

We deployed our file replication system on the Electrical Engineering Computer Group (EECG) network at the University of Toronto. The network connection between the computers in this network operates at 100Mbps.

To test various features of our system we deployed the file replication system on 11 UltraSparc machines running Solaris operating system. One of the machines serves as a bootstrap node so that all nodes joining the network have a single node to connect to initially. This was required as for UDP based connections it is necessary to know a specific port and an Internet address of at least one node in the replication system to join the system.

Once all nodes joined the system we created 1000 files in the system. We then filled those files with random content. The file sizes were uniformly distributed between 0 and 32kB. We then waited for the system to settle to ensure that all files in the system were replicated exactly the same number of times. The number of replicas per file was determined by the replication factor.

Once the system was settled we performed several tests to determine how well our system operates. These tests are described in the next section.

5.2 Testing Methodology

We tested our system to determine several key parameters. These parameters were: read performance, network traffic due to replication when a node fails, time to recover from a node failure and how resilient our system is to node failures.

To test read performance we issued 1000 requests, 1 read request from each node in the system per second. This effectively put a load of 660 messages per minute on the network, which caused some of the messages to be dropped by the network due to congestion. We repeated the same experiment for replication factors of 1 through 4.

We also measured the network traffic observed by the system due to replication messages generated by each node when a node fails and leaves a system. We combined the measurements of network traffic with the measurements of time it takes the system to stabilize. We observed the network traffic due to replication in the system and waited until it has been reduced to zero. The network traffic generated in this experiment was only due to replication messages. We repeated the experiment for replication factors of 2 through 5.

Finally, we measured how resilient our file replication system is to node failures. The resilience was measured by how effective the system is in retrieving files once a node has failed. We varied the replication factor from 1 to 3. After the system has stabilized we killed one of the nodes, randomly, and waited for the system to recover from node failure. Once the system recovered from node failure, we sent 1000 read request for files that should be in the system and counted how many requests failed.

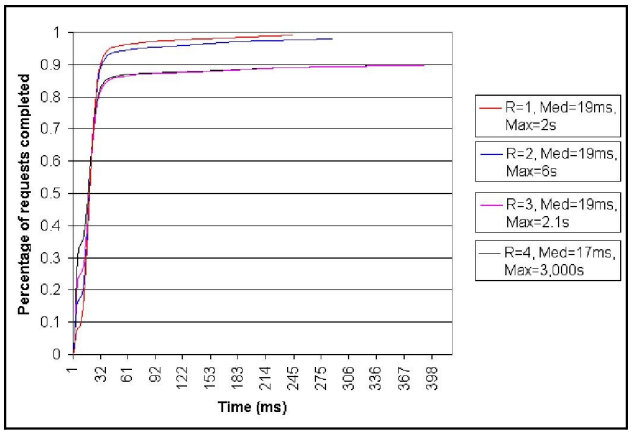

Figure 4: Read Performance for various replication factors

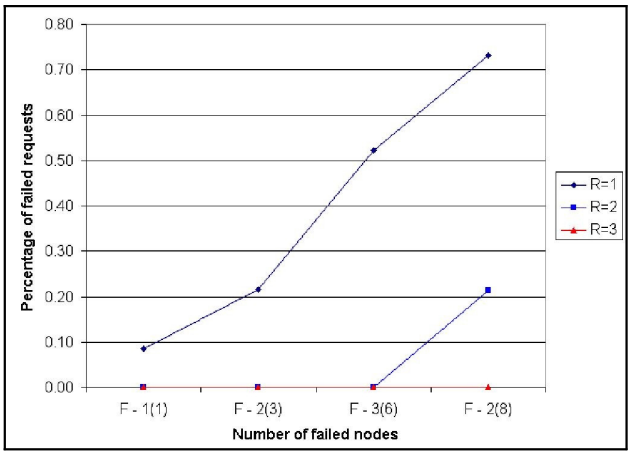

Figure 5: Fault resilience results

Once this test was complete there were only 10 functioning nodes left in the system. We then failed another two nodes, then another three and another two nodes again., measuring the read success rate after each set of failures.

We discuss the results of our experiments in the following section.

5.3 Discussion

The tests described showed how our system behaves in various circumstances and validated our design. We now discuss the results of these tests and comment on any observations we made during these tests. We will first look at read performance, then network traffic and time to recovery after a node failure. We finish our discussion with the discussion of resilience test results.

5.3.1 Read Performance

Read performance results are presented in Figure 4. The graph represents a cumulative distribution function (CDF) of the percentage of reads processed within a specified time. The Y axis shows the fraction of reads that were completed in the time specified by X axis.

For all replication factors the median read response time is approximately the same. In fact, for replication factors 1 through 3 the median response time is 19ms. Only when the replication factor is at 4 we see a small reduction in median response time, down to 17ms. The improvement in median response time is expected as more nodes hold a copy of the file thus it takes fewer node hops to find a file.

Interestingly, there is no evidence of change in median response time for replication factors of 1 through 3. We attribute this to three factors. First, we subjected the network to a high load of read, thus a number of packets were dropped during transit and had to be resent after 30 seconds. This caused many messages passed by Pastry from one node to another to be interleaved with user requests, delaying their processing.

The second reason for these results is that a number of packets were delayed more than 30 seconds, as issuing 1000 read requests prevented the resend queue from resending messages on time. This is evident by a very large maximum request processing time (over 3000s). Since this is a problem of the user interface that blocked the resend queue operation, we disregard excessively long response times as we know the actual time they were issued was not the same as the time they were marked as being issued.

Finally, it becomes evident in this test that the size of the messages sent during periodic updates takes its toll. We expect that if instead of sending full updates between nodes, we send short queries that would return a notification if an update is needed, the network traffic would be much lower, allowing messages to be processed faster. The modification to reduce network traffic during periodic updates is left as future work.

5.3.2 Node Failure Recovery

The second test we performed aimed at measuring the network traffic that occurs when a node fails. Concurrently, we measured how much time it takes for the system to recover from a node failure. The results of this test are shown in Figure 5.

In Figure 5 the time it takes for a system to recover and the network traffic generated within the system are plotted against the replication factor. We can see that time to recovery remains about the same, in the vicinity of 180 seconds. It slightly increases for higher replication factors due to more messages being passed within the system. The network traffic on the other hand grows approximately exponentially from 2MBytes to about 50MBytes.

Similarly to results in section 5.3.1, we see that the frequent periodic updates cost us in recovery time. This is because the number of messages grows linearly with the replication factor, but many messages are dropped. This causes many more messages to appear in the system due to periodic updates.

5.3.3 Failure Resilience

The final test focused on how resilient our system is to

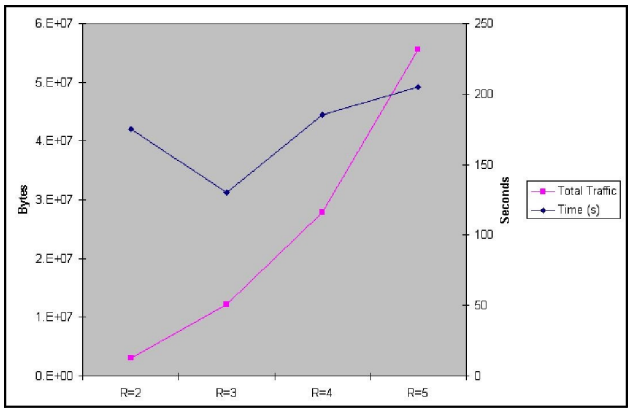

Figure 6: Network traffic and recovery time after a single node failure

node failures. We define resilience as the ability of the system to maintain all of the data despite node failures. To measure resilience, we tested the percentage of failed requests due to non-existent files when a node in the system fails for different replication factors. The results are shown in Figure 6.

Figure 6 shows we see plots, one for each replication factor 1 through 3. As expected, the percentage of failed nodes grows linearly with the number of failed nodes for replication factor of 1.

For replication factors 2 and 3, we see that no files were lost until 8 nodes failed. It is possible for files to be lost when 2 nodes fail simultaneously (F-2), however this was not the case in our test. When we fail another 3 nodes simultaneously, we see that replication factor of 2 was insufficient and about 21% of the requests failed. For replication factor of 3 all files were maintain.

This test shows that our system is resilient to node failures and is able to maintain majority of files even when several nodes fail simultaneously.

6. Conclusion

In this report a distributed file replication system based on Pastry DHT was presented. Our system is implemented as an extension of Pastry and was deployed in a real environment.

In the course of this project we have gained significant knowledge about distributed systems, specifically how to handle failures and how important they are. While our system design is relatively robust, significant amount of work was placed on making it resilient to node failures. We put emphasis on cases of simultaneous node failures. This is shown well in our results, where the system responds well to multiple concurrent node failures.

The project in this course was an enjoyable experience as well as very educational. Implementing this project provided us with insight into distributed system design. We use this knowledge to propose some future changes that could be applied to our work.

7. Future Work

In this work we noticed that there are several improvements that could be adopted to enhance the performance of our system.

The first improvement involves reducing the overhead of periodic updates. Currently, periodic updates send all contents of a file from one node to another. However, it is possible that while the update replica message arrives without a problem, the acknowledgment does not. In this case our system resends an update message during periodic update, including all contents of the file. This can produce unnecessarily large network traffic. Instead, we could adopt a strategy to send a query from one replication node to another, asking if a file update is needed. This would greatly reduce network traffic, especially for updating large files.

The second improvement addresses the problems we faced with the resend queue. Currently, the resend queue can be blocked by user commands, which affects the performance of our system. Alternatively, we could implement the resend queue to work concurrently with the replication node, preventing it from being stalled.

The third improvement we are considering is using version vectors to detect inconsistencies in the system. Using version vector would allow a user application to specify a reconciliation policy. This would make our replication system more attractive from end-user point of view.

Finally, to test our system would like to run it on PlanetLab. PlanetLab provides a large network of machines that we could test our system on. This would provide us with an opportunity to study how scalable our system is.

8. References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. OSDI 2002

[2] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, SIGCOMM, 2001

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In Proc. ACM SIGCOMM'01, San Diego, CA, Aug. 2001.

[5] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In Proc. SOSP, p. 188-201, October 2001

[6] Microsoft Corporation, Microsoft Windows 2000

Server Documentation: Understanding Active Directory, [Online Document, Available HTTP]: http://www.microsoft.com/windows2000/en/server/help/sag_adintro_10.htm?id=273

[7] Microsoft Corporation, Microsoft Windows 2000 Server Documentation: Understanding the Active Directory schema, [Online Document, Available HTTP]: http://www.microsoft.com/windows2000/en/server/help/sag_ADschema_Understand.htm?id=304

[8] Microsoft Corporation, MSDN Library: Platform SDK: Active Directory, [Online Document, Available HTTP]: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ad/ad/what_is_the_active_directory_replication_model.asp

[9] S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. IEEE Transactions on Software Engineering, SE-9(3), p.240-247, May 1983.

[10] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems,” Proc. IFIP/ACM Middleware, Heidelberg, Germany, November 2001.

[11] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica, “Toward a Common API for Structured Peer-to-Peer Overlays” author = "F. Dabek and B. Zhao and P. Druschel and I. Stoica", In IPTPS '03, Berkeley, CA, February 2003.