FAULT TOLERANCE IN STRUCTURED PEER TO PEER NETWORKS

by

Malayil Philip George

A thesis submitted to the

Graduate School - New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

And approved by

New Brunswick, New Jersey

May, 2006

ABSTRACT OF THE THESIS

Fault Tolerance In Structured Peer To Peer Networks

By Malayil Philip George

Thesis Director : Dr. Manish Parashar

Structured Peer to Peer Networks has the advantage of being able to provide guaranteed lookups in bounded time, something, which unstructured peer to peer networks cannot provide. However, this advantage comes at the price of an increase in the cost of maintenance and reliable operation. The basic problem is that peers may join and leave the network at will. When either a join or leave operation occurs, then the network needs to be restructured so as to still continue correct operation. This problem is compounded by nodes that leave without informing the network of its intention to depart (node failure) and further compounded by multiple node failures (possible partitioning of the network). This thesis looks into these problems and provides effective methods to counter them. Heartbeat signals are used to detect node failures, and in such an event a reorganization message is sent across the network to bring back the proper appropriate structure. In the case of multiple node failures, a method has been proposed to let the network partition into multiple structured networks and then later fuse these smaller networks into the whole network again. Instead of propagating the node failure information across the network, efforts are made to only update the closest nodes in the structure. It is shown that such a method does not adversely affect lookup times. These methods have been written in Java for operation over the JXTA overlay network and tested on the same.

Acknowledgments

I would like to thank my family for their support during my studies in graduate school. I am grateful to my advisor Professor Manish Parashar for his invaluable guidance, encouragement and support throughout my stay at Rutgers. I would also like to thank my committee members, Dr. Ivan Marsic and Dr. Yangyong Zhang for their advice and help during the thesis study.

I would also like to thank Vincent Matossian for developing METEOR which formed the underlying framework used in this thesis. Thanks to the CAIP computer facility staff, who helped with quickly resolving issues on CAIP's computers.

Table Of Contents

Chapter 1
INTRODUCTION

There are several milestones in human evolution that has made us the dominant species on the planet - speech, communication and documentation. The ability to communicate effectively over distances has made civilizations successful. They could be smoke signals of the native Indians or tom-toms of African tribes. In the modern day, the ability to communicate swiftly and effectively is even more critical, be it, for personal reasons or multi-million dollar transactions. The ubiquity of computers has led to networks being built that connect these devices around the world. It is essential that these networks remain stable, robust and tolerant to faults, so that faults in a particular area does not bring the entire network and underlying communication infrastructure down to a standstill. The objective of this thesis has been to analyze the fault tolerant nature of a structured peer-to-peer network, CHORD, and to improve it's robustness to faults in METEOR. The primary aspects of stability in METEOR that have been looked into are Bootstrapping, stabilization in the face of node joins/failures and the establishment of alternative routing paths.

Computer Networks gained prominence and relevance by following a server-client architecture. In this model, a node called the server, hosted data and served this data and other functions to nodes known as clients. The clients connected to the server when they were in need of data or other services, and the server provided it to the clients at that time. This type of operation has a number of advantages, the primary one, being speed of operation. On the other hand, a primary disadvantage with this model is that there is a central point of failure - the server. The server can be replicated to increase uptime, but the possibility of failure still exists. Another disadvantage, is that the server needs to generally have a lot of resources at it's disposal - bandwidth, computing power etc., as these servers typically have to service thousands of clients simultaneously.

In an effort to improve the fault tolerance and load balancing of the server client model, alternative networking paradigms are constantly being investigated. One of these models, known as Peer To Peer (or P2P) networking, is fast gaining ground. These networks, are for the most part self-organizing and, remove the distinction between client and server. Instead, each node in the system are equal peers and simultaneously function as both client and server. These nodes communicate amongst each other to provide data and services to other nodes or peers. This model has the advantage of being decentralized and hence being more resilient to failure of a single or set of nodes. Also, load balancing is inherently provided as operation is distributed among all the nodes that form the network.

There are primarily two kinds of P2P networks - unstructured and structured. In an unstructured P2P network, peers form random connections between each other and use these connections to find data or request services from other nodes in the network. This lack of structure makes it difficult to pin down responsibility to any particular individual or entity for the network, along with making them resilient to network failures and to a certain degree self-healing. It however takes away properties of speed and guaranteed lookups - something that structured networks like Chord, Pastry, CAN and Tapestry hope to provide.

In a structured P2P network, the nodes are arranged in a definite predetermined order, one that forms a ring in the case of CHORD. This structure helps in bounding the number of messages needed to find an item in the network and can also provide guarantees on the existence of the object in the network.

Applications of a structured P2P network

i) Remote backup

Data can be tagged with an identifier by a user and then provided to a group of nodes that offer back up services. These peers decide amongst themselves who would be responsible for storing this item based on the identifier and allocate the object to that peer. Later, the user could retrieve the object based on the identifier. When the network is again presented with the identifier, it determines who has the object, retrieves it and returns it to the user. These operations are totally transparent to the user, who only needs to present an object to the network to store.

ii) Cooperative Mirroring

Content can be mirrored across multiple nodes to provide a form of load balancing. The data blocks can be hashed onto hosts and this enables the distribution of data evenly among hosts.

iii) VOIP

A structured overlay can be formed and used to route voice packets between any two locations in the world using peers that dynamically join and leave the network. This could lead to cost-effective long distance telecommunication around the world.

These applications require that the underlying network remains resilient to faults, especially when the nodes that form the network are transient in the network. This thesis goes into providing mechanisms for making such a structured network stable.

Architecture

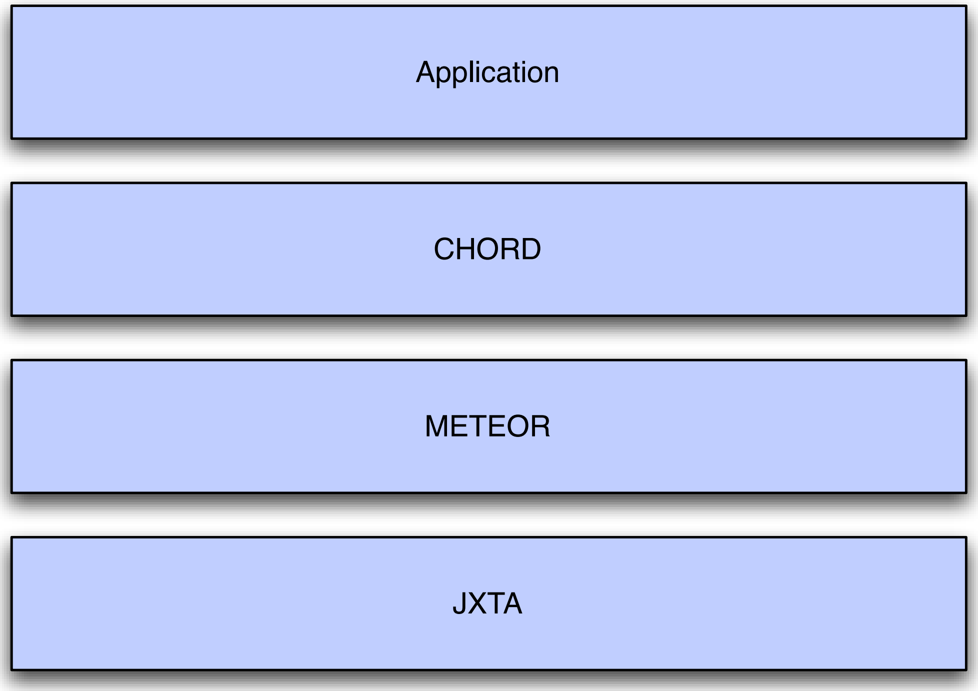


Figure 1

The above figure depicts the architecture of the implementation of the structured P2P network used in this thesis. CHORD is run on METEOR, which is in turn operated over the JXTA P2P framework.

Overview of the Thesis

This thesis presents methods to stabilize the implementation of CHORD in METEOR and improve it’s resilience to node failures and other topological changes in the network. The following sections describe

RELATED WORK

work done in the field that complements this thesis study

CHORD

in detail the underlying P2P network structure used

JXTA

the framework used to build the P2P network

MECHANISMS TO INCREASE METEORS FAULT TOLERANCE

identifies potential pitfalls in METEOR and addresses them

RESULTS

CONCLUSION

Chapter 2

RELATED WORK

In order to provide fault tolerance in P2P networks like Chord, other implementations have looked into replicating the services on multiple nodes. Essentially, more than one node becomes responsible for a key in the id-space. Hence, if a node wants to send a message to a node covering a point *z* in the id-space, it could contact one of several different nodes that cover *z*. This method also helps in controlling spam. There could be the possibility of a node generating arbitrarily false data items, for a query. But, with replication, when a node wants to pass a message to a node that is responsible for a point *a* in the id-space, it contacts all the nodes that are responsible for the point *a* in the space. In each time step, a node receives a message from a multitude of nodes in the previous time step. It passes on the message to the next node in the path, only if it receives messages from a majority of the nodes in the previous step. A disadvantage with this method is that if a point in the id-space is covered by O(log n) nodes, then a lookup takes $O(\log^2 n)$ messages. The lookup time, however, still takes O(log n), as O(log n) messages are passed in *parallel* at each step.

Another fault tolerant algorithm is described by *Athica Muralitharan, Seth Gilbert* and *Robert Morris* in “Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data”. This method also uses replication to provide fault tolerance. It replicated different objects over different nodes in the system. Each such set of replicated nodes is known as a configuration. Any read or write operation to an object would require the cooperation of the entire configuration. Reconfigurations are allowed, so as to provide correct operation against node failures/joins. The nodes in a configuration are a preset number of successors that follow an object in the id-space. The immediate successor is known as the primary. If two or more nodes believe that they should be the primary for a particular object, then a re-configuration process takes place using the Paxos protocol to ensure consensus. Eventually, consensus is reached, and the read/write operation is allowed only after this.

The basic CHORD protocol suggests that each topological change be propagated around all nodes that might be affected around the ring. While this approach might work when the number of failures are few, it doesn't scale well with a large number of failures.

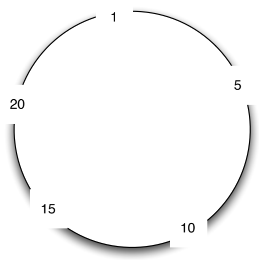
While replication is a way to provide for high availability, there are other kinds of faults that may occur, such as a network partition. Also, with replication, the number of resources needed to maintain the network increases, as does the number of messages needed to ensure a correct lookup or other network operation. In an effort to work around these, this thesis looks into improving the stability and fault tolerant nature of METEOR without the added overhead of replication.

Chapter 3
CHORD

Chord at it's simplest is a distributed lookup service, which when implemented in a network, maps a key to a node. It does this in an effort to make data location, one of the key concerns of P2P networks, extremely efficient. Data location is implemented by assigning keys to data and then these keys to nodes. To locate the data, we just need to do a lookup on the key and find the node that is responsible for that key.

The Ring

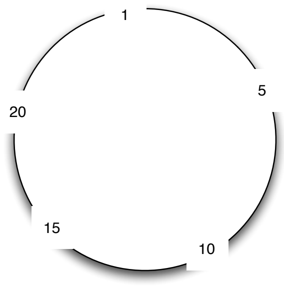
Chord arranges it's nodes into a ring. Each of these nodes are assigned an id (chosen at random) that is drawn from the same id-space as the keys that are allocated to these nodes. This id allocation is done by consistent hashing, a hashing mechanism that helps ensure that, every node in the network is roughly responsible for the same number of keys. A key difference between consistent hashing, as presented in paper (xx), and Chord, is that consistent hashing assumes that every node knows about the majority of the nodes in the network. Chord, on the other hand, needs to know only about the node immediately following it on the ring for correct operation. This node is known as the successor.



Theoretically, each node needs to know only about it's successor to ensure proper lookups and operation of the network. In order to maintain it's position in the ring accurately, i.e to know about it's correct successor at all times, every node also keeps track of the node which immediately precedes it in the ring. This node is known as the predecessor.

A simple lookup method is to forward the query on the successor ring until it arrives at the node which owns the key being queried. This, however, is highly inefficient with a worst case running time of O(n). To improve on this lookup time, each node also maintains knowledge of 'd' other nodes, where, d is the dimension of the ring. The dimension of the ring is log(N) in a N node network. This list of 'd' nodes is known as the finger table, the formulation of which is explained below.

Each node maintains a list of nodes in a table whose size is equal to the dimension of the ring. The i^{th} entry in the table contains the id of the node which owns the key which is 2^i away from it's own node id. This means the first entry in the table always contains the node that owns the key which is 2^0 away from the current node id, i.e. the successor of the node. The fourth entry contains the node which is 2^4 away from the current node id. A direct result of the finger table structure is that a node knows more about the nodes closer to it on the ring than the ones farther away.   An example finger table for a node in a 4 dimension network is given below.



Armed with a finger table, key lookups at a node are reduced with a high probability to O(log N). When a node wants to lookup a key, it checks it's finger tables entries. If the key is present in the finger table it can contact the successor to the key directly. If not, it finds the key that is closest to the queried key and forwards the lookup to the successor of that key. The query is handled in a similar fashion at that node and in this way the query makes it around the ring until it reaches the target node. In the example network above, if node 3 wants to contact the node that holds the key 14, it contacts 11 which is the closest node to 14 it has on it's finger table. 11 then looks up it's finger table and repeats the process, which is iterated at every hop until the target node is reached.

Node Joins

There are two invariants that Chord needs to maintain at all times in order to ensure correct lookups. These are -

1. Every node is aware of it’s correct successor
2. All keys are held correctly by successor(key)

To maintain these invariants, Chord needs to perform the following operations with each node join

- The joining node needs to set it’s successor, predecessor and fingers correctly.
- Existing nodes need to update their fingers and predecessors to reflect the addition of the new node.
- Keys that the new node is responsible for need to be transferred to it.

When a new node starts up, it needs to find a existing node on the Chord ring by some bootstrap mechanism. Once it does this, it can query that node to find it’s successor and predecessor on the ring. It then does lookups to identify the successors of all the entries in it’s finger table. In the worst case, this is of the order O(m logN) in a m dimension N node network. To optimize this, a node can check up if the successor to a finger is actually different from what it has previously looked up before initializing a query.

Some of the existing nodes on the ring will need to have their finger tables updated to reflect the addition of the new node in the ring. This can either happen during stabilization or by messages from the new node to indicate the change in state. The number of nodes that need to be updated is with a high probability of the O(log N). Since, a newly joining node can be responsible only for keys that are currently owned by it’s successor, it only needs to contact that node for the transfer. The amount of data that actually needs to be transferred would depend upon the application that is running on top of CHORD. This could be anything like cooperative mirroring, time shared storage, distributed indexes or large-scale combinatorial search.

Stabilization

In order for lookups to succeed, it is essential that every chord node knows who it's correct successor is and to have a reasonably accurate finger table. When a node joins the network, it sets out to discover it's successor and create a finger table. It announces it's presence to it's successor, but, the rest of the network remain unaware of it's entry. In order to make every node that should reflect the newly joined node in it's finger table aware of the node, they periodically run a stabilize routine.

During the stabilize routine, each node 'n' asks it's successor 's' for it's predecessor 'p'. It then determines whether that predecessor 'p' should be it's successor. This might happen if 'p' just entered the network and the other nodes are not yet fully aware of it's presence. If so, it updates it's successor entry with 'p'. It also informs 'p' of it's existence, so that, 'p' can update it's predecessor entry with 'n'. 'p' updates it's predecessor entry with 'n' only if it doesn't know of any closer predecessor than 'n'. If it does know of a closer predecessor, then it could inform 'n' of a possible better choice for it's successor. This gives 'n' the opportunity to establish proper successor-predecessor relationship with that or a better node.

All nodes also periodically run a routine to check it's finger table for accuracy and to fix discrepancies. It is with this or a similar routine that newly joining nodes create their finger tables and existing nodes update their finger tables to accommodate newly joined nodes. Each node also checks to see if it's predecessor is up. If the predecessor is down, it clears it predecessor's pointer and accepts notify messages from other nodes to set predecessors. It also queries it's successor in an attempt to find a better predecessor across the ring.

Chapter 4

JXTA

JXTA (or juxtapose) is an open source peer-to-peer framework developed by Sun Microsystems. It is primarily built over JAVA but a C interface is also available. As it's open source it can be ported to any modern computer language. It allows for a range of devices, from cellphones to fully loaded servers to join the network, and allows for these devices to communicate in a decentralized manner. JXTA peers form an overlay network and allow for peers to join the network even if they are behind a firewall or use different network transports. Every resource in JXTA is identified by a 160-bit URN known as the jxta-id. This allows a peer to change it's IP but still maintain a constant jxta-id.

JXTA Peers

There are two main types of JXTA peers - edge peers and super peers. Edge peers are generally the nodes that are transient in the network or have limited resources. These form the general bulk of peers that join the network for a particular service for a short period of time, and then, leave when they are done. Super peers on the other hand are special dedicated peers, have a lot of resources at their disposal and are long lived in the network. Again, there are two main types of super peers - Relay peers and Rendezvous peers.

Relay peers form connections between peers behind a firewall or NAT, and allows these peers to join the jxta network. It does this by using a protocol that can traverse the firewall (like HTTP). Rendezvous peers are used for discovering resources in the network. These resources could be peers, communication channels or any other service that could be of interest to other peers. It also provides for message propagation among peers that are split among subnets. In this case, there has to be a rendezvous peer in each subnet. Peers that want to communicate across subnets, send messages to the rendezvous server. The rendezvous servers pass messages between each other, thereby, enabling communication across the subnet. Any peer in the JXTA framework can become a rendezvous server, as long as it has the necessary credentials and resources.

Advertisements

Resource discovery in JXTA takes place through a system of advertisements. An advertisement is a XML document that describes any resource in the network - peers, pipes, groups, services etc. When a peer joins the network, it sends an advertisement to the rendezvous server and other local peers, containing it's jxta-id and indicating it's existence in the network. Similarly, a group of peers that provide similar functionality can issue a group advertisement indicating the existence of that group. Advertisements are cached at each peer and the rendezvous server, and they time out after a while.

Pipes

Pipes are virtual communication channels in JXTA and messages are transmitted between peers using these pipes. There are three main types of pipes - Unicast, Unicast secure and Propagate. A unicast pipe is a one to one connection and connects a peer with another. In contrast, a propagate pipe is one to many and can be used to transmit messages from one peer to many other peers. Pipes in JXTA are unidirectional and asynchronous. To obtain bi-directional connectivity between two peers, we would generally have to use two uni-directional pipes. A bi-directional pipe service class in the Java implementation of JXTA provides this abstraction. Any means of reliability have to be built into the application as JXTA pipes are unreliable.

With this framework, JXTA provides a mechanism for peers to join a network, advertise it's presence, define common functionality and form a group, and provide these services to any peer joining the group. METEOR is an implementation of a DHT built over JXTA and primarily provides CHORD and CAN implementations of a structured P2P network. The objective of this thesis has been to study the resilience of METEOR to faults and improve it's ability to withstand large-scale topological changes in the network.

Chapter 5
STABILITY IN METEOR

Meteor, designed and developed at the TASSL lab at CAIP is an implementation of DHT's using JXTA on java. JXTA is a open set of protocols that enable any device on a network to communicate with another in a P2P manner. When a node joins the JXTA network, it becomes a JXTA peer. JXTA peers together form a virtual network, where every peer can access every other peer including it's resources, regardless of whether it is behind a firewall or not. JXTA peers can form new groups or join existing ones. A group is generally bound by a common framework or working ground. This enables us to form new P2P groups that run a particular protocol. Meteor has been written to run a Chord or CAN DHT on a set of JXTA peers. Much work has been done in an effort to stabilize Meteor and extend it's capabilities to work against faulty or malicious behavior. We will go over the changes and extensions made in subsequent sections.

Bootstrapping

This is a minor change that makes the Chord peers search for a pipe advertisement instead of a peer advertisement when it starts up. With the prior implementation, when a peer running Chord starts up, it sends out a advertisement for all the peers that have joined the Chord group. It does this by contacting what is known as a rendezvous server, and a broadcast on it's local network. The rendezvous server sends out the request to other rendezvous servers that it is connected and also queries the networks it is connected to. When a peer that has joined the Chord group replies, it sends out the advertisements of all the peers it knows of which is running Chord. These may be the advertisements of peers that has contacted it before. However, there is no guarantee that all the peers that these advertisements belong to are still running i.e the peer that is broadcasting these advertisements has no way of restricting the broadcast to peers that are still active. If the peer that is currently joining the network, picks an advertisement that corresponds to a peer that is no longer running, it blocks forever, waiting for that peer to reply. This becomes especially apparent, when we take down the entire network or most of it, and then start it up again. This is a quirk of JXTA and does not reflect normal network operation.

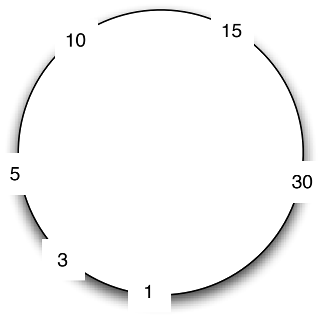
A workaround to this problem is to load the peers with a pipe advertisement. A pipe is a mechanism by which peers send messages to each other in JXTA. A pipe may be an input pipe (a peer accepts messages on this pipe), an output pipe (a peer sends messages out on this pipe), or an abstract interface of a bi-directional pipe (which performs both input and output pipes, by creating two pipes). When the peer starts up, it sends out a message on this pipe. All JXTA peers in the Chord group are listening on this pipe and reply to the bootstrap request message. So, now, when a node picks an advertisement from one of the replies, it knows for sure that the node is still running with a high probability. There is a small off-chance that the particular node that sends out a reply goes down after replying. This can be worked around by iterating through the replies that has been received.

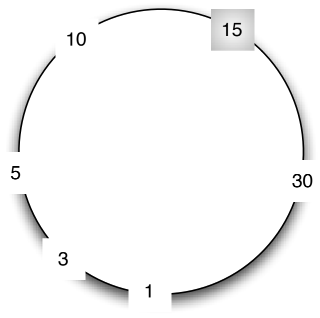
Stabilization

Peers join, they also leave. This is a simple inalienable truth common to any P2P network. They enter the network for some purpose, which may be to download some content or to perform some computations over the network. Once they are done with this, they typically leave, unless they are given some further incentive to stay on in the network. A network that is to remain stable over an extended period of time, has to adapt to these transient peers. Peers especially, have to modify their routing tables in minimum time to reflect these changes in peer states across the network.

A possible solution is to have each node, on joining or leaving, calculate the positions of all nodes that may point to it in the Chord ring, and then tell these nodes to update it's finger tables. However, with nodes rapidly joining and leaving the system it may not be possible to keep updating the finger tables of all other nodes accurately.

Another solution is to lazily update the finger tables of nodes while keeping the predecessor and successor of the nodes accurate at all times, or correcting these with minimal delay. This works just as fine (illustrated in the following example) , without much loss in routing efficiency, as keeping the finger tables accurate at all times.





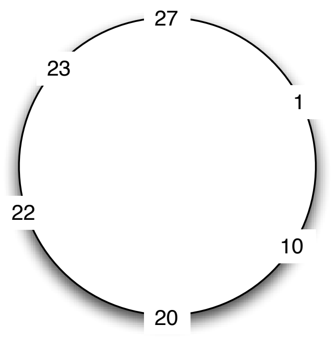
In the above Figures, let the left figure indicate the initial stable state of the chord ring. If node 5 wanted to find the successor of say key 27, it would look up it's finger table and see that the closest successor to 27 that it has is 15. So, it queries 15 and asks it to find the successor to key 27. 15 looks up it's finger table and sends the query to 30, which replies that it is the owner of the key in question. The number of hops is 2 in this case which is of the order log(n).

Now, let us take the case of the second figure, where we assume node 15 has crashed or otherwise left the chord ring prematurely. Now, when 5 wants to find the successor of 27, it looks up it's finger table and again sees 15 as the closest successor. It then tries to contact 15, but in this case times out. It then contacts 10 which is it's successor and is still up. 10 looks up it's finger table and sees 30 is the closest successor and queries it. 30 replies saying it is the owner of the key in question. The number of hops in this case is 3 which is still of the order log(n). So, there is hardly any loss of efficiency in routing as long as the successor's of each node is correct. In the worst case, if all the fingers of most nodes go down, then the efficiency would reduce to O(n). However, in a later iteration of fix fingers, the table will be fixed and efficiency would return to

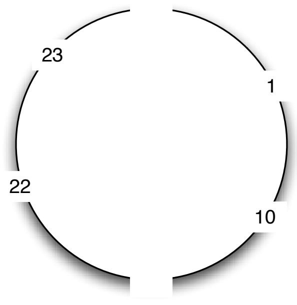
O(log(n)). The reduction of efficiency to n is unavoidable in the case of large scale node failures, but, is only a temporary condition.

Stabilization with Multiple Rings

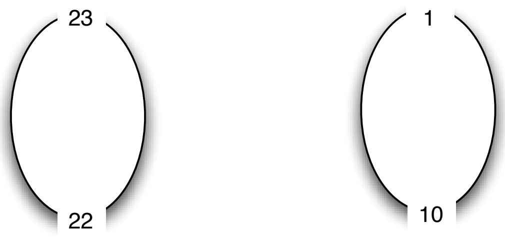
There are occasions, which are fairly common in small rings, where multiple node failures results in nodes having no knowledge of certain sections of the ring. This is, because, each node is aware of it's successor, predecessor and the nodes in it's finger table. The nodes in the finger table are typically ones that are not too far away from it in the ring. Hence, if nodes on the farther edge of it's finger table were to go down, this node would have no idea about the existence of nodes beyond that or on how to reach them. A quick solution to this problem, is to let nodes form multiple closed rings around the edges of it's finger table, and have each of these rings elect a leader. These leaders advertise their presence by a broadcast mechanism. The presence of multiple advertisements indicates the need to join more than one ring. The leaders communicate with each other to bring this about and is illustrated in the following example.



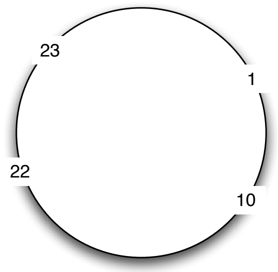
Let the above figure reflect the initial stable state of a sample chord ring. If nodes 27 and 20 were to go down at about the same time, a bridge of sorts is formed across the ring.



With our current implementation of detecting node failures, node 22 would detect that it’s predecessor 20 has gone down, and node 1 would detect the failure of node 27. Both 22 and 1, now will try to find their current predecessors in the ring. 22’s request would make it’s way to node 23, whose finger table will not have any indication of 1 in a network of this size. Since, 23 has no other node to forward the request to, it will instruct 22 to set 23 as predecessor, while it sets 22 as it’s successor. In a similar fashion, 10 would instruct 1 to set it’s predecessor to 10 while it sets it successor to 1. Now, we have two separate chord rings as show below.



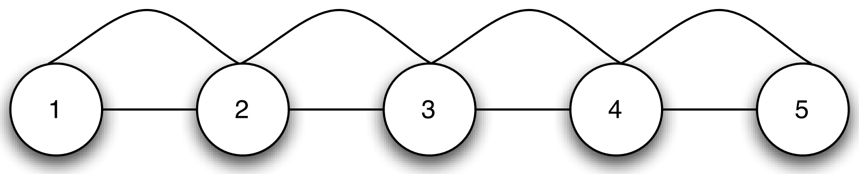
These two rings now elect the node with the lowest node id as their leaders. The leaders periodically broadcast their presence on the network. When they receive the broadcast of another leader, they contact that node to merge the two rings into one. This is a simple process, as only the end nodes in each ring need to update their successors and predecessors to find their correct position in the new ring. In the above example, 1 and 22 would be elected leaders. When 1 receives 22's broadcast message it asks 22 for the predecessor of 1 in 22's ring and the successor of 10 in the ring. Here, 23 and 22 would reply to these find successors and predecessors respectively, resulting in 1 and 10 updating their predecessors and successors. We would now have the ring formed correctly as below.



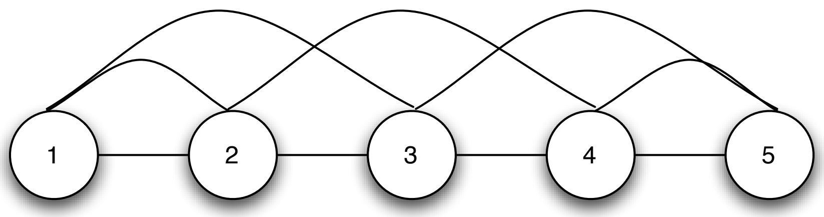
This method is very cost effective as it has minimum number of adjustments to form a newly stabilized ring even in the presence of large scale node failures or widespread network outages. However, it makes the assumption of the availability of a broadcast mechanism. In the absence of such a mechanism, we could employ the services of the bootstrap server or some other well known server. Instead of broadcasting their presence at regular intervals, the leaders would report their presence to the bootstrap (well known server). If this server were to receive multiple advertisements, it would inform the senders of the advertisements of each others presence, so that they can merge together into a single ring.

Mutually Exclusive Routing

Forwarding messages in a secure manner in the presence of failures or malicious behavior requires the establishment of alternative routes, as we have seen in a previous section. This is in the hope that one of the alternative routes will exclude the node that has failed and the message will then with a high probability make it's way from the source to the destination. Establishing these mutually exclusive routes, is however, not a simple task and could take a lot of effort even in a structured P2P network. If we were to look into Chord, each node is connected to a node ahead of it and behind it. This is the extent of knowledge of the network it is required to have to operate correctly.

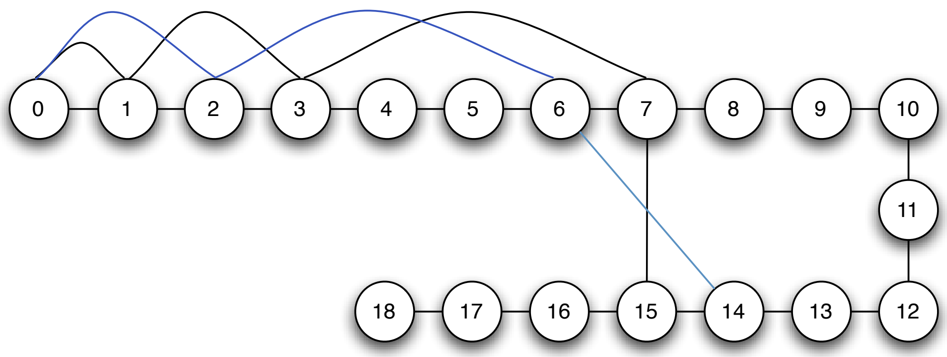


If we were to take the above topology where every node is aware of only it's adjacent nodes, and the messages were to follow this path, it is impossible to develop a mutually exclusive route for the messages to follow. This is because, each message can only go one step ahead towards the destination at every hop. We can then try to make each node aware of it's two adjacent nodes.



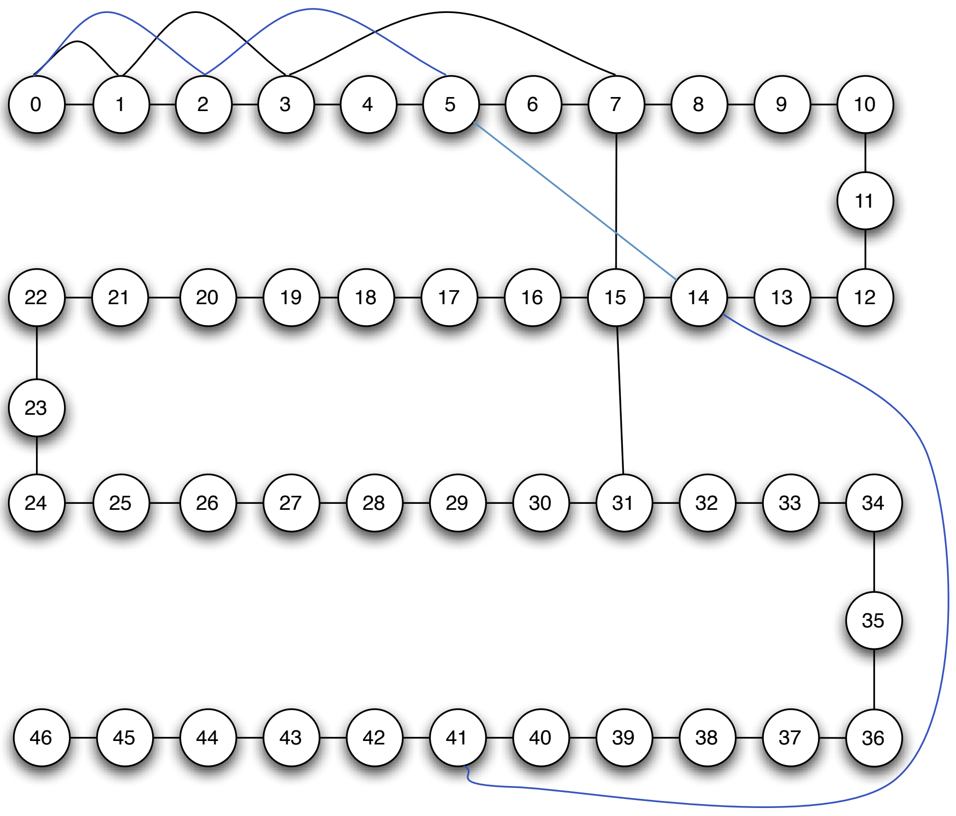
Now, in this case there are exactly two mutually exclusive routes. If node 1 wanted to send a message to node 5, then it has the option of sending it via path

1-2-4-5 or 1-3-5. This linear hop count, scales linearly towards the number of possible mutually exclusive routes i.e a hop count of 3 would result in 3 mutually exclusive routes and so on, up-to n/2 where n is the number of nodes in the ring. However, the message forwarding in this case is inefficient as it takes an O(n) hops to forward a message in a network with n nodes. In fact, it was because of this inefficiency that finger tables were introduced in the chord ring, so as to bring the routing efficiency to the O(log(n)).



The above figure depicts a scenario where node 0 attempts to generate mutually exclusive routes with the help of it's finger table. It asks it's successor and the node following that to generate two routes to the target node. As long as both nodes keep extending the hops by a power of 2 at every hop, it would seem that a mutually exclusive route is being produced. However, a closer look would reveal that this is not necessarily the case. The above example works when the ring is full i.e. every key is owned by a separate node. However, this is not necessarily (more often not) the case. The same node that owns the key 7 might also own the key to 6 and similarly for 14 and 15. Hence, a collision is taking place right there. This is because, the distance of separation between the hops is just one key when we use only a power 2 table.

To avoid the above problem, we could use a combination of a power 2 and power 3 table. This would, however, entail expanding the chord ID space to a power-3 dimension. In this case, the distance between successive hops grows rapidly and there is a greater chance of generating mutually exclusive routes.



As can be seen above, the distance between keys grows rapidly when we use a combination of a power 2 finger table and a power 3 finger table. This gives a greater possibility of generating a mutually exclusive route, although it is still impossible guaranteeing.

A mutually exclusive route can however be guaranteed if we use a successor path for one message and a predecessor path for the second message. This would with a high probability cause the message to be delivered in the presence of failures as long as there is no collusion between the nodes. In order to detect failed nodes, we could route in a iterative fashion as described in a previous section, and route around the failed nodes.

Chapter 6

RESULTS

JXTA provides a ready made framework to implement and deploy a P2P network, with minimal resources needed by a peer. However, to do this efficiently, requires an understanding of the underlying JXTA model of networking, which as can be seen provides it's own challenges. Circumventing these issues and exploiting the features of JXTA optimally to provide a fault tolerant CHORD ring has formed the basis of this thesis. The major issues have been in

- Bootstrapping
- Stabilization in the face of topological changes
- Establishing Mutually Exclusive Routes

Bootstrapping

For a peer to join a CHORD ring, it needs to determine it's position in the ring and establish it's presence. It does so by contacting another peer already in the ring, which is known as the bootstrap peer. Making this contact efficiently, within the constraints of JXTA has been the major issue with this step.

Let us assume the time taken to contact and receive a response from a node to be equal to 't' seconds. If we were to use peer advertisements to identify a bootstrap peer, then the best case response time would be 't' seconds. But, since peer advertisements are cached we may be faced with the issue of having to iterate through dead nodes. If there are 'n' such dead nodes, then the time taken to find a live bootstrap peer would be 'tn' seconds.

In contrast, with our modified method of using pipe advertisements, the time taken to contact a live node in the worst case is only ‘t’ seconds. This is because only live nodes respond to messages in the pipe and there is no issue of resolving advertisements of nodes that have otherwise left the network.

Stabilization In The Face Of Topological Changes

This step is important, so as to provide proper lookups in the face of failures and maintain the guarantees provided by CHORD. The problem in maintaining accurate finger tables at all times is that there are potentially ‘d’ peers that point to a particular peer via it’s finger table. If finger tables are to be kept accurate, then the number of queries that needs to be propagated on detecting a failure is ‘d’ messages, if a peer goes down. In case ‘n’ peers go down then the number of messages that need to be propagated is ‘nd’. Resetting successors requires an additional ‘n’ queries. The total number of queries is n(d+1).

If we were to relax the constraints on the finger table being accurate at all times, then we need to only send out the ‘n’ queries to maintain correct successors. As shown, this still lets us maintain O(log n) lookups as long as the finger tables are reasonably accurate. The extra (d+1) messages can be sent at staggered times as to ease the load on the network. When the underlying network is unstable, and this is causing node failures, then this reduction in the number of queries could turn out to be crucial in keeping the network up.

Large scale node failure could also be due to the loss of a communication channel between ends of a network. By allowing the rings to partition and then rejoin when the communication channel comes back up, we allow the services to continue on subsets of the network. An example scenario could be a chat application. At times users may be cut off from other users in a network application. But, they can still continue talking on the rings they are connected to, and when these rings are brought together, resume chatting with the world at large.

Mutually Exclusive Routing

Setting up mutually exclusive routes is essential to route around failed nodes, in the transient periods between network instability and network stability. Such a route can be established using the predecessor part of the network, with an additional cost in terms of maintaining the predecessor finger table.

Chapter 7
CONCLUSION

There is still a lot of work to be done before structured P2P networks come out in the mainstream market. However, there is a lot of potential for these networks in various fields such as data storage, multimedia sharing and content delivery. As diverse as it's applications may be, each of them require secure operation for it's stability. This thesis paper has gone into the various aspects of security that may afflict these networks, and has proposed solutions to some of them. Some of them such as secure message forwarding using the predecessor ring, as well as expanding the chord ring to support power 3 tables have been implemented in Meteor. The stability of the ring in Meteor has also been improved upon with some of the techniques described. However, much work still needs to be done, which includes but is not limited to secure bootstrapping and peer authentication.

References:

1. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, “CHORD: A Scalable Peer-to-peer Lookup Protocol for Internet Applications”, Feb. 2003, IEEE/ACM Transactions on Networking

2. Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron and Dan. S. Wallach, “Secure Routing for structured peer-to-peer overlay networks”, Dec. 2002, 5th Usenix Symposium on Operating Systems Design and Implementation

3. Todd Sundsted, “The Practice of Peer-to-peer computing: Trust and Security in peer-to-peer networks”, July 2001, IBM developerWorks library

4. Anjali Gupta, Barbara Liskov, Rodrigo Rodrigues, “One Hop lookups for peer-to-peer overlays”, USENIX Assosciation HotOS IX

5. Vasilieos Vlachos, Stephanos Androutsellis-Theotokis, Diomidis Spinellis, “Security Applications of Peer-to-peer networks”, Computer Networks,45:195-205, 2004

6. David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy, “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”, 1997, ACM Symposium on Theory of Computing

7. Anjali Gupta, Barbara Liskov and Rodrigo Rodrigues, “Efficient Routing for peer-to-peer overlays”, 2004, First Symposium on Networked Design and Implementation

8. David Evans, Chenxi Wang, Jun Xie, “A payment mechanism for publish-subscribe systems”

9. M. Pease, R. Shostak, and L. Lamport, “Reaching Agreement in the presence of faults”, 1980, Journal of the association for Computing Machinery

10. Alan Mislove, Peter Druschel, “Providing Administrative Control and Autonomy in Structured Peer-to-peer overlays”

11. Emil Sit, Robert Morris, “Security Considerations for Peer-to-peer Distributed Hash tables”

12. John R. Douceur, “The Sybil Attack”

13. Steven M. Bellovin, “Security Aspects of Napster and Gnutella”