VRIJE UNIVERSITEIT

MASTER'S THESIS

Fault Tolerant Rings: Creation and Maintenance

Author:
Atul MEHTA

Supervisor:
Prof. Wan FOKKINK

Second Reader:
Dr. Paolo COSTA

October 2, 2008

Contents

List of Figures

Abstract

Numerous algorithms have been provided for maintaining the topology of a ring network in a fault-free environment. But a simple, fast and effective fault-tolerant algorithm has been lacking in this field so far. In this report, we seek to address this problem and provide a detailed design to solve the problem in the general case. As our detailed analysis proves, our algorithms function very well in providing a high degree of fault tolerance at very little additional cost. We further construct a new peer-to-peer system built using our ideas. This new system provides very fast routing and a high degree of fault tolerance at almost no additional cost. We also propose a new geometric solution to the problem which provides a reasonably high degree of fault tolerance.

1 Problem Statement

Many structured peer-to-peer systems have been built using the ring topology as a base, for example `Symphony` and `Skipnet`. But the most prominent of such systems is undoubtedly `Chord` [10]. The success of this kind of systems has led to further investigation on how to maintain the topology of a network in the face of concurrent leaves and joins. Different systems adopt different approaches, but in the ultimate analysis a simple and symmetric solution was lacking thus far. In real-time we need to factor in not only concurrent joins and leaves, but also the fact that faults are liable to arise and that too frequently. A fault could arise from the sudden crash of a node (either hardware/software crash) or if the network connecting that particular node gets disrupted. But this cannot be the only definition of a fault. In general, there is no method to distinguish a slow and unresponsive system from a system that has crashed. Also an involuntary leave from the system can also be construed as a fault. While some crashed systems recover in a finite time (by rebooting the system) we cannot say this is true in every single case.

All these factors come into play significantly when we attack the problem of maintaining the topology of the network in a practical real-world scenario. It is obvious that any system which departs from its set topology, will suffer from significant performance degradation. A partitioning of the system for example can leave the majority of users without access to important data. In this chapter we discuss existing work done in this area, and elaborate on the leave/join protocols discussed in [3]. *We focus solely on a fault-free environment in this chapter, and will proceed to the more practical scenario of dealing with faulty environments later on.* The problem of designing an adequate and robust, topology maintenance protocol for the *fault-free* environment has been satisfactorily dealt with by Li, Misra and Plaxton in [3]. In this chapter, we explain the essence of the various protocols they have developed for maintaining the topology, in various types of rings, such as unidirectional, bidirectional and P2P rings. Extending their solution to *faulty* environments is however not an easy task and has, in fact, been posed as an open problem in [3].

1.1 Current Approaches to Topology Maintenance

While there is no standard approach to the problem of topology maintenance, we can still classify existing attempts into two broad categories, a *passive* approach and an *active* one. In the former, neighbor variables are *not* instantaneously updated to reflect the node joins or leaves. A background protocol is run periodically to restore the topology to the desired state. The obvious drawback of this method is the cost of running such a background task in terms of computing and network resources and also the latency introduced. The effect of changes in the network are not immediately obvious to the other nodes. The *active* approach stresses that all neighbor variables get updated immediately at every change in the network (joins and leaves). This method is costly as every node has to keep track of any change in its neighbors, wasting time and resources on this task. The *active* approach, on the other hand, implicitly assumes that any departure from the set topology is detrimental, which is not always the case. Most systems can tolerate temporary departures from the ideal topology to a significant extent.

Protocols for coping with joins and leaves can fall in either of the two categories. For example

`Skipgraphs` handles joins and leaves actively, while some others handle joins actively but leaves passively, like `Tapestry`. The correctness proofs for some of the join/leave protocols, which are typically used, are not rigorous but are operational and sketchy, as in [10].

1.2 Basic Definitions

We consider an *asynchronous* network with a fixed and finite set of nodes (processes), denoted by V. Elements denoted by *u, v* and *w* represent processes belonging to the set V. Each processes has left and right neighbors (as in the case of a bidirectional network). Variables l and r denote the neighbor variables. For example $u.r$ denotes the right neighbor of *u*, while $u.l$ is the left neighbor. A process is called an "*x* process" iff $u.x \neq nil$, where a "*nil*" process is one that does not belong to the set V and consequently no neighbor variables are applicable to it. V' denotes the set of processes, $V \cup \{nil\}$. Thus, it is easily seen that neighbor variables like l and r belong to the set V'. Message delivery is reliable and asynchronous, meaning that messages take a finite but indeterminable time to reach their destination. We do not make any assumptions regarding the order of the delivery. Now we are ready to define the unidirectional ring.

$\mathbf{ring(x)} =< \forall u,v : u.x \neq nil \wedge v.x \neq nil : path^+(u,v,x) >$,
where $path^+(u,v,x) =< \exists i : i > 0 : u.x^i = v >$ and where $u.x^i = u.x.x.x \ldots x$ with *x* repeated *i* times.

Thus, *ring(l)* and *ring(r)* denote unidirectional rings with a left and right channel respectively. Combining the two, we can arrive at the definition for a bidirectional ring.

$$\begin{aligned} biring(x,y) = \quad & ring(x) \wedge ring(y) \\ & \wedge < \forall u : u.x \neq nil : u.x.y = u > \\ & \wedge < \forall u : u.y \neq nil : u.y.x = u > \end{aligned}$$

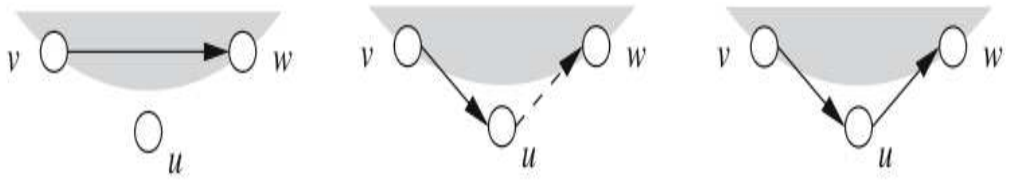
Thus *biring(l, r)* denotes a bidirectional ring.

Now that we have defined the ring, we concentrate on the protocols to be used. We will use Gouda's Abstract Protocol Notation [1], and assume that an execution of the protocol consists of an infinite sequence of actions, where each action is a series of atomic steps (send/receive or local assignments). The subsequent results can be seen to hold for *sequential executions* of the protocol, one where steps of each action are contiguous. It can also be proved that the results hold for an *interleaving execution*, i.e. one in which steps belonging to different actions may be interleaved. In the following sections, we list the protocols to be adopted for joins and leaves for the unidirectional and bidirectional ring, respectively. Finally we combine the join and leave protocol to come up with a combined protocol.

1.3 The Unidirectional Ring

First let us look into the case of node joins for a unidirectional ring. Assume a process *u* wants to join *ring(r)* (i.e. every node in the ring has a right neighbor). Process *u* invokes a function, $contact()$ which returns an existing suitable node *v* for joining. If *ring(r)* is empty it returns the id of the calling node. itself. Figure 1.1(a) depicts the initial topology of the network. Process *u* sends a $join()$ message to *v*. Node *v* accordingly changes its right pointer to point to its new neighbor *u*. Process *v* also sends a $grant(w)$ message to *u* so that it can update its right pointer accordingly. This is depicted in figure 1.1(b).

Upon receipt of the $grant(w)$ message from *v*, process *u* updates its right variable to point to *w*. Node *u* is now part of the network (figure 1.1(c)). The flow of messages during the course of the join protocol is depicted in figure 1.2.



(a) Initial topology (b) After the *join*() message (c) Successful completion of join

Figure 1.1: Joining a unidirectional ring [3]

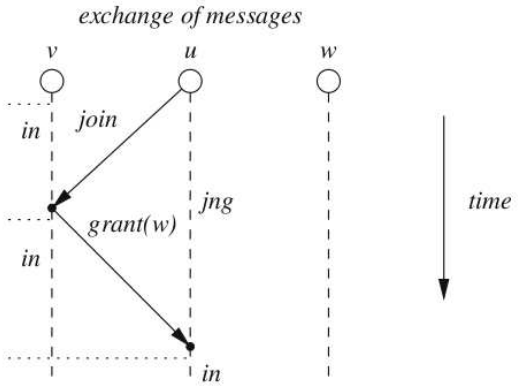


Figure 1.2: Exchanges of messages during the join protocol [3]

Now we are ready to list the *join* protocol. Note that an action in our protocol is represented by a sequence of statements. A statement can be of any of the three forms, a local assignment, sending a message or a selection. The first two are obvious. The selection statement is of the form ***if*** ⟨ *branch* ⟩ ***fi***. Each of the branches is represented in the form ⟨ *local guard* ⟩ → ⟨ *statement list* ⟩ and different branches are separated by ▷.

```
process p
var {in, out, jng}; r : V'; a: V'
init s = out ∧ r = nil
begin
        s = out → a := contact();
           if a = p → r,s := p, in
            ▷ a ≠ p → s := jng; send join() to a fi
        ▷ rcv join() from q →
           if s = in → send grant(r) to q; r := q
            ▷ s ≠ in → send retry() to q fi
        ▷ rcv grant(a) from q → r,s := a, in
        ▷ rcv retry() from q → s := out
end
```

If a node sends a *join()* message to another node which is currently not part of the network (for instance, it might have just left), then it receives a *retry()* message. In such a case, we would expect the initial node to run *contact()* again. A fresh node id would be received and the process can start all over again. While [3] does not list a specific *leave* protocol for a unidirectional ring, we can develop on the concepts listed in that paper, and come up with a *leave* protocol. We have to keep in mind though, that leaves in a unidirectional ring involve a high cost. Consider the case in figure 1.1(c). If now process *u*

wants to leave the network, it has to inform process v so that we can achieve $v.r = w$. For this a message has to be routed around the ring, since the network is unidirectional. If the size of the network is N, then the number of hops to reach the predecessor of any given process is $N-1$. The *leave* protocol proceeds as follows. Process u routes a message, *leave(w)*, to its predecessor v informing its intent to leave. v will now set its right variable to point to w, and this will remove u from the ring. The leave protocol for the unidirectional ring can now be stated as follows. Note that the function **predecessor(*u*)** will route the message to some node v such that $v.r = u$.

```
process p
var {in, out, lvg}; r : V';
init all process states are either in or out;
        the in processes form ring(r) ;
        t = nil ∧ (s = in ⇒ r = nil)
begin
        s = in →
          if r = p → r,s := nil,out
           ▷ r ≠ p → send leave(r) to predecessor(p); r,s,t := nil,lvg,r fi
      ▷ rcv leave(a) from q →
          if s = in ∧ r = q → r = a ; send ack(nil) to q ;
           ▷ s ≠ in ∨ r ≠ q → send retry() to q fi
      ▷ rcv retry() from q → s,r,t := in,t,nil
      ▷ rcv ack(nil) from q → s,t := out,nil
end
```

1.4 The Bidirectional Ring

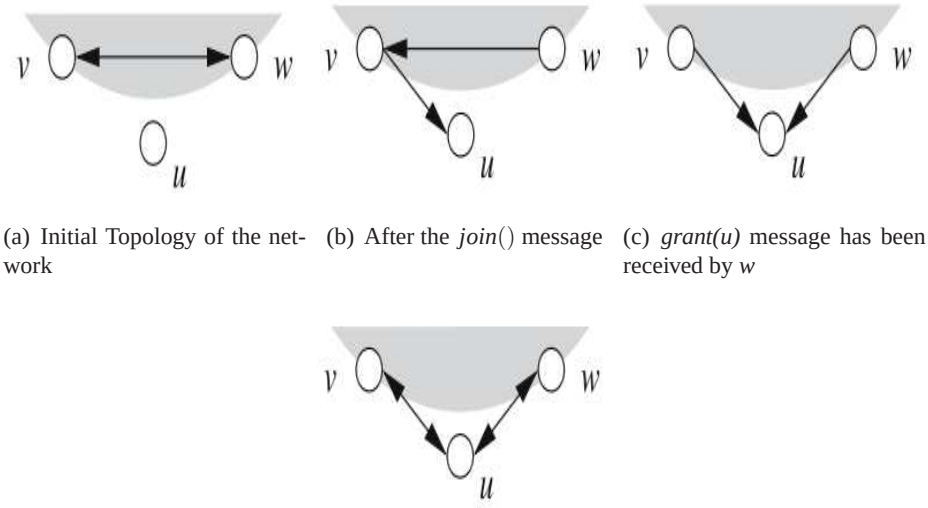
Devising suitable protocols for joins and leaves for the bidirectional ring, involves more work than for the corresponding unidirectional case. An important feature of the leave and join protocols is that they should be symmetric to each other. This is needed if we are to combine them effectively later on.

1.4.1 Join Protocol

We first approach the question of devising a protocol for *joins* for a bidirectional ring *biring(l,r)*. The initial state of the network is depicted in figure 1.3(a). Node u sends a *join()* message to v, which will in turn adjust its right pointer accordingly ($v.r = u$), figure 1.3(b).

Process v also sends a $grant(u)$ message to its previous right neighbor w, which adjusts its left pointer accordingly as shown in figure 1.3(c). Finally process u will set its right and left pointers to w and v respectively, and join the ring. The new topology is reflected in figure 1.3(d). Four messages are exchanged during the course of the join process, and this is shown in figure 1.4.

The $join()$ protocol is listed below.



(a) Initial Topology of the network

(b) After the *join*() message

(c) *grant(u)* message has been received by *w*

(d) Successful completion of protocol

Figure 1.3: Joining a bidirectional ring [3]

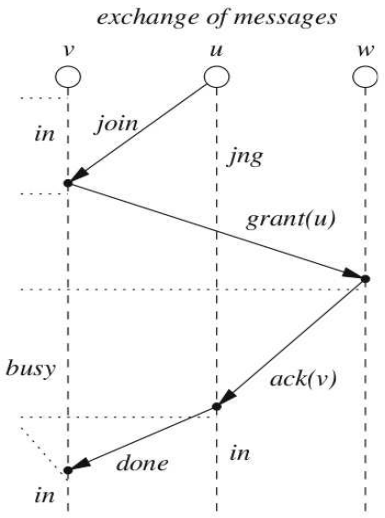
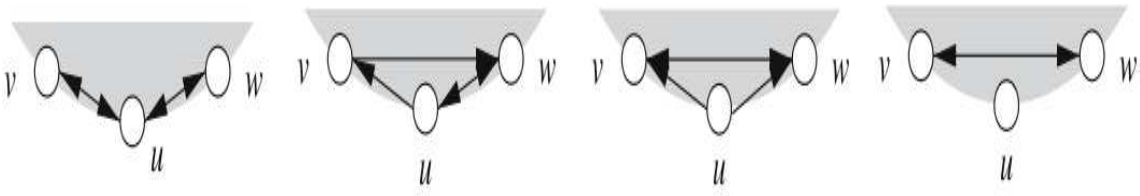


Figure 1.4: Exchanges of messages during the *join* protocol for the bidirectional ring [3]

```
process p
var {in, out, jng, busy}; r,l : V'; t,a: V'
init s = out ∧ r =l =t = nil
begin
        s = out → a := contact() ;
          if → r,l,s := p,p,in
           ▷ s ≠ p → s := jng; send join()to a fi
      ▷ rcv join() from q →
          if s = in → send grant(q) to r;
            r,s,t := q,busy,r
           ▷ s ≠ in → send retry() to q fi
      ▷ rcv grant(a) from q →
           send ack(l) to a; l := a
      ▷ rcv ack(a) from q →
           r,l,s := q,a,in send done() to l
      ▷ rcv done() from q → s,t := in,nil
      ▷ rcv retry() from q → s := out
end
```

1.4.2 Leave Protocol

We now turn our attention to designing the *leave* protocol. As mentioned earlier, we strive to make both the *join* and *leave* protocols highly symmetrical to each other, so as to combine them later on, in one protocol. The initial state of the network is depicted in figure 1.5(a). Process *u* wants to leave the network and it informs its left neighbor of its intention to leave. Process *v* accordingly changes its right pointer, figure 1.5(b) and contacts its new neighbor informing it of the change.



(a) Initial Topology of the network (b) After the *leave*(*w*) message (c) *grant(u)* message has been received by *w* (d) Successful completion of the join

Figure 1.5: Joining a bidirectional ring [3]

Process *w* changes its left pointer accordingly and informs *u* that it can leave (figure 1.5(c)). Process *u* can now remove itself from the network; the changed topology is depicted in figure 1.5(d). Similar to the previous *join* protocol, the *leave* protocol also uses four messages. The flow of messages during the *leave* protocol is depicted in figure 1.6.

The complete *leave* protocol is listed below.

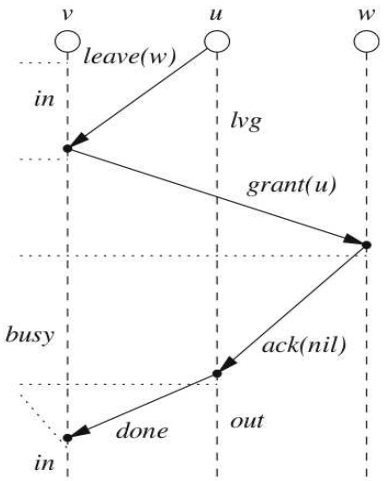


Figure 1.6: Exchanges of messages during the *join* protocol for the bidirectional ring [3]

```
process p
var s: {in, out, lvg, busy}; r,l : V'; t,a : V'
init all process states are either in or out ;
        the in processes form biring(r, l) ;
        t = nil ∧ (s = out ⇒ r = l = nil)
begin
        s = in →
          if l = p → r,l,s := nil,nil,out
           ▷ l ≠ p → s := lvg; send leave(r) to l fi
      ▷ rcv leave(a) from q →
          if s = in ∧ r = q → send grant(q) to a;
            r,s,t := a,busy,r
           ▷ s ≠ in ∨ r ≠ q → send retry() to q fi
      ▷ rcv grant(a) from q →
          send ack(nil) to a; l := q
      ▷ rcv ack(a) from q →
          send done() to l; r,l,s := nil,nil,out
      ▷ rcv done() from q → s,t := in,nil
      ▷ rcv retry() from q → s := in
end
```

1.5 The Combined Protocol

A combined protocol which handles both joins and leaves is easy to make now. This is because both the *join* and *leave* protocol, listed in the earlier sections, are symmetrical to each other. To arrive at an integrated protocol, we just combine the two. The new protocol is listed below.

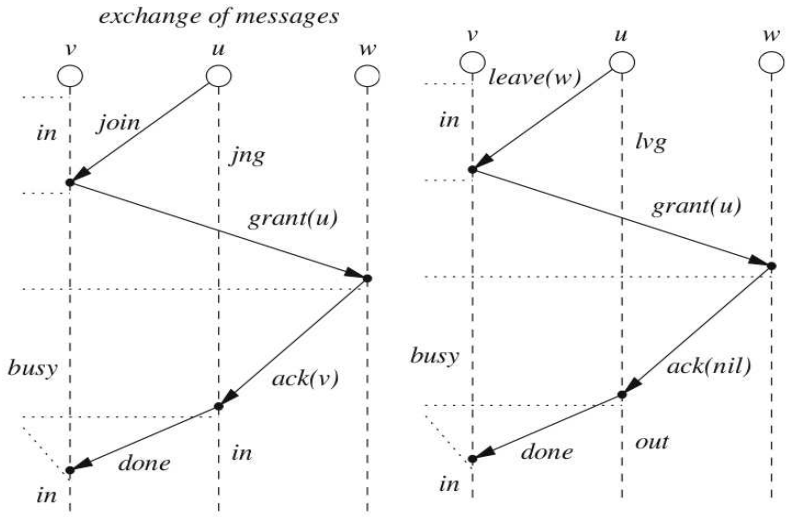
```
process p
var s: {in, out, jng, lvg, busy}; r,l: V'; t,a: V'
init s = out ∧ r = l =t = nil
begin
        s = out → a := contact();
          if a = p → r,l,s := p,p,in
           ▷ a ≠ p → s := jng; send join() to a fi
       ▷ s = in →
          if l = p → r,l,s := nil,nil,out
           ▷ l ≠ p → s := lvg; send leave(r) to l fi
       rcv join() from q →
          if s = in → send grant(q) to r;
            r,s,t := q,busy,r
           ▷ s ≠ in → send retry() to q fi
       rcv leave(a) from q →
          if s = in ∧ r = q → send grant(q) to a;
            r,s,t := a,busy,r
           ▷ s ≠ in ∨ r ≠ q → send retry() to q fi
       rcv grant(a) from q →
          if l = q → send ack(l) to a; l := a;
           ▷ l ≠ q → send ack(nil) to a; l := q; fi

       rcv ack(a) from q →
          if s = jng → r,l,s := q,a,in
            send done() to l
           ▷ s = lvg → send done() to l;
            r,l,s := nil,nil,out fi
       rcv done() from q → s,t := in,nil
       rcv retry() from q →
          if s = jng → s := out
           ▷ s = lvg → s := in fi
end
```

1.6 Conclusion

The protocols in the earlier sections are explained in depth in [3]. Detailed assertional proofs are also provided there. Variants of the protocols are also listed, which handle joins/leaves in FIFO networks and for P2P networks like `Chord`. But on further analysis it soon becomes apparent that these protocols do not deliver, when we seek to apply them to faulty environments. In a real-world scenario, processes can crash arbitrarily, messages can get lost, or nodes can become so slow and unresponsive that it becomes difficult to tell them apart from crashed ones. The protocols that we have so far elaborated on do *not* have any provisions to handle these kind of errors and will fail in such a faulty environment. We list a few examples that serve to illustrate this point further.

Figure 1.6 illustrates the messages exchanged during the course of a *leave* and *join* in a bidirectional ring. Note that in both cases four messages need to be sent for the successful completion of the protocol. Let us consider the case of the *join*, as illustrated in figure 1.7(a). If subsequent to the initial *join()* message, any one of the remaining three messages fail to get delivered, then the network is left in an inconsistent state. For example, if the *ack(v)* message fails, then both nodes *v* and *w* will point to *u*, which will make process *u* behave like a **blackhole**. Process *u* will then be able to receive messages, but cannot contact any of its neighbors on the ring, since it is still waiting for *ack()* to arrive. If the *grant(u)* fails, then the ring gets broken between *v* and *w*. Process *v* will not be able to contact *w*, since its right neighbor is process *u*, which is out of the ring currently. It is important to bear in mind that the

non-delivery of messages is *not* only because of faulty links. If that were the case, we could always keep retransmitting the messages, and soon the protocol would get completed. But what if any of the nodes u, v or w crashes during the course of the protocol? The messages never arrive and the protocol fails, leaving the ring in an inconsistent state.



(a) *Join*-ing a bidirectional ring (b) *Leave*-ing a bidirectional ring

Figure 1.7: Exchange of messages during *join/leave* [3]

Let us take a look at the *leave* protocol now, as depicted in figure 1.7(b). The situation is no better than the previous case. If immediately after receipt of the *leave(w)* message, *v* crashes, then the ring gets broken at that point. Now *u* will never be able to leave the ring, as the *grant*(u) will not be forthcoming. If node *u* crashes before sending the *done*, then *v* will be condemned to live in the *busy* state, or if the *ack(nil)* fails, then both *u* and *v* have to suffer. An obvious solution could be to implement a series of *timeout*'s. This has a two-fold disadvantage. Firstly, it robs the above protocols of their generality by making the network essentially synchronous. Secondly, if a process has become busy or temporarily disconnected at that particular time, then the system of timeout's will fail. A timeout system might not be able to make a distinction between a slow process and a crashed one. After expiry of the timeout, other processes will assume that the process in question has crashed, which need not always be the case. *This is thus the central problem that we face, namely modifying the Li-Misra-Plaxton protocols* [3] *to cope with faults in an asynchronous network.*

An elegant solution to this problem, albeit for a *synchronous system*, exists in [8]. The basic idea is to adopt the *"Paxos Commit Algorithm"* to provide fault tolerance. It uses three stages in every join/leave operation. In the first, participating processes determine the action to be undertaken, the second phase prepares the action, and the last phase commits it. The method used is flexible also, in the sense that instead of the Paxos commit, we can use the common 3-phase commit protocol also. The concept remains the same. But the main drawback of this algorithm is its reliance on a synchronous network.

A promising line of investigation appears to be that when a process suspects some other process to have crashed, then it simply bypasses it and contacts another suitable node in the ring. For example, assume that process *v* has crashed in figure 1.3(b). Node *u* then contacts the left neighbor of *v*, and along with *w*, *u* will join and repair the ring. While this method is promising, an element of centralization gets introduced. How can a node "by-pass" crashed/slow nodes? It has to know the identity of the neighboring nodes, but for this some central server might have to keep information about the current topology of the network. This is both impractical and undesirable. Even a group of servers distributed at various parts of the ring would be impractical, as no guarantee can be provided on the *freshness* of the information

they store, especially in networks with high churn. There would be extra work involved in keeping all the servers up-to-date.

Note that while fault tolerance might not be important to the working of distributed rings created over the Internet, it is vital factor in the design of P2P networks. Without reasonably good fault tolerance, performance of P2P networks suffer. There is another aspect of the Li-Misra protocols that is worth mentioning. *They do not provide a progress property*. For example, if a process desires to leave the network, it should eventually be able to do so. This can be provided only by a *progress* property.

In the remaining part of this thesis, we will propose a de-centralized solution to the fault-tolerance problem. Our solution works in networks with high churn, and can accommodate not only rings, but also P2P networks based on the ring topology, like `Chord`. We will also present a new design for creating highly fault tolerant and fast ring networks, assuming certain *apriori* information about the size of the ring is made available. Additionally both our solutions also guarantee the progress property, which is lacking in [3]. Lastly we present a partial geometric solution with interesting characteristics. This will provide a high degree of fault tolerance.

2
A General Solution

We approach the problem from a different direction. Let us assume that there is a group of *successor* and *predecessor* functions, denoted by $f()$ and $g()$ respectively. The domain and range of these functions cover the whole space of possible node id's. So if node id's are denoted by 160 bits, then both the domain and range are equal to $[0, 2^{160} - 1]$. The behavior of the *successor* and *predecessor* functions for normal rings (uni- and bidirectional) and *"wrap around"* rings[1] like `Chord` is defined below.

> $f()$ applied to an existing node id gives the id of the *successor* of the node in the ring and likewise $g()$ gives the node id of the *predecessor* of that particular node. $f()$ applied successively n times, will give the node id's of the first, second, third and so on till the n^{th} successor of that particular node. The same holds for $g()$. We assume that a value of ε denotes error, i.e. the node id does not exist or in case of wrap around rings, the true successor is different.

For a bidirectional ring, it is important that the functions $f()$ and $g()$ be inter-related, i.e. for some node with an id x, the following holds always $x = f(g(x))$ and $x = g(f(x))$. In other words, $f()$ and $g()$ are the inverses of each other. For unidirectional rings, we can do away with this restriction. Only one set of functions $f()$ and $g()$ will need to be applied, depending on the direction.

2.1 Suitable Functions

What could be suitable group of functions which can be used for $f()$ and $g()$? Our functions will necessarily have to be scalable and easy to compute if they are to be deployed in large-scale distributed systems. Also a regular function like $f(x) = x + 1$ is not feasible, as it will amount to pre-ordering the node id's. Pre-ordering the id's can lead to grave security issues. If an adversary is able to predict the id, for each node about to join the network, he can initiate man-in-the-middle attacks easily. But the main problem with pre-ordering is the uneven nature of the lookups that it causes. This point is further detailed in section 2.4. On the other hand, symmetric encryption functions are highly suited to our requirements and do not suffer from any of these pitfalls.

If we assume that symmetric encryption functions are cyclic, then it would necessarily have a very large period T such that for some x, $f(x) = f(x+T)$[2]. A large period is necessary as the ring cannot

[1] In a ring network like `Chord`, the successor of a node has a numerically larger id than its own. This is important if we have to order them around the identifier circle. But there would exist some node, whose successor has a numerically smaller id than its own. If this does not hold, we will end up with a straight chain of processes with increasing id's. Thus, the set of nodes "wrap around" to form a circle.

[2] If the function is periodic, then sooner or later, we arrive at a cyclic arrangement of nodes. In essence, we get the "wrap around" feature. While this is strictly not necessary, it is desirable. Currently it is unknown whether the symmetric encryption functions are cyclic or not. In case they are cyclic, then definitely the period would be very large. In case they are proved to be acyclic, we would then, need to re-phrase the meaning of "period" as applied to our successor and predecessor functions. In such a scenario, the period would represent the maximum number of processes that can be supported in the ring. Processes with id greater than the maximum would get mapped to some existing process.

grow beyond size T. For example, functions like $f(x) = (ax)\ mod\ N$, where a, N are large prime numbers and N is smaller than the absolute range, would be unsuitable. In such a case, the ring cannot have more nodes than N, after which duplicate id's start to creep in. Symmetric encryption functions are easily invertible if one knows the shared, secret key. Thus in our scheme, we initially employ $f(x)$ equal to $K(x)$ and $g(x) = K^{-1}(x)$, where $K()$ is the symmetric encryption function. This is an important choice, especially if we are implementing bidirectional rings, because we now have a set of readily invertible functions. Symmetric encryption functions are also reasonably fast to compute on modern machines. There is no distinct pre-order among the node id's if we are using symmetric encryption functions. If an adversary is unaware of the id of the starting node and the shared secret key being used, he won't be able to predict the id's of the successive nodes. The node id's will appear just as a set of random numbers. Because of these desirable properties, symmetric encryption functions are very suitable for implementing the successor and predecessor functions.

2.2 Methodology

We use the successor and predecessor functions to construct our ring. Let us assume for the time being, that we are constructing a bidirectional ring. Figure 2.1 illustrates how the successor function comes into play in defining the ring.

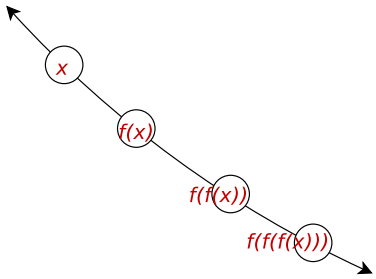


Figure 2.1: A portion of a bidirectional ring

Here we make a crucial assumption. *We assume that every node can resolve the IP or the physical address of every other node as long as it knows its process id in the network.* This can be arranged by a suitable external mechanism. For example, every node joining the ring will post its physical address and node id to a central server or a group of servers. The latter is of course more preferable. We need to further elaborate on this point since we intend to use this assumption in the section relating to P2P systems also. The main pitfall here is in, introducing an element of centralization. This has to be avoided or else it will render the entire scheme impracticable. Instead of resorting to a central group of fixed servers, we can instead use a dynamic set of already existing nodes (alternatively a set of "beacon" nodes can be built into the system). This small group of nodes holds the physical and virtual address information, of all the nodes participating in the system. The "beacon" nodes need not have a fixed membership. Instead we can make it dynamic and any change in the membership in the set of "beacon" nodes is transmitted to all the participating nodes in the system (using either flooding or gossiping). Moreover, the *(physical, virtual)* address information can be widely replicated among the set of the "beacon" nodes or it can alternatively be broken down amongst these nodes. In the latter method, the virtual address space is broken up and divided equally among the set of "beacon" nodes. Overall, our scheme should sufficiently guard us centralization.

Note that our scheme is very different from storing or replicating the topology of the network, since we, only store the node id and its physical address. Moreover they can be stored in any order and do

not need to have one-to-one correspondence with the actual pattern of joins and leaves in the network. Nodes which are leaving the network may perhaps want to remove their id from the database. While this is preferable (in case of rings) it is not necessary. It can be seen that with this limited information one cannot replicate the topology of the network. Any participating process can query such a database with the node id as the key and obtain the physical address of the process corresponding to that particular node id (if it exists). An empty string ε signifies non-existence of the node id searched for. Overall our assumption is not totally unreasonable and can be implemented with little extra effort. Once the physical address is resolved, then the processes can contact each other directly.

Theorem 2.2.1. *Given any pair of arbitrary **live** processes in the network. These processes will always be able to contact each other in a finite number of steps, as long as none of the two processes crash or depart from the network in that particular time interval.*

Proof. Given a starting process x, all the other process id's are of the form $f^n(x)$ where $n \in N$, the set of natural numbers. Without any loss of generality, let us denote the two live processes as p_i and p_j where $i < j$. The node id's corresponding to p_i and p_j are thus $f^i(x)$ and $f^j(x)$ respectively. Let the period of the function $f()$ be denoted by T. Consider the following two cases.

- *Unidirectional Rings* For p_i to contact p_j it keeps resolving $f^k(f^i(x))$, where k is 1 initially and increased at every iteration. When $k = j - i$, p_i is able to successfully resolve the id of node p_j. If node p_j desires to contact p_i, we apply a similar procedure. When $k = T + i - j$, p_j is able to successfully resolve and contact p_i.
- *Bidirectional Rings* With bidirectional rings, we are able to move both ways. p_i can contact p_j by applying the procedure mentioned above in $j - i$ steps. For p_j to contact p_i we use the inverse function $f^{-1}()$ (basically the function $g()$). This is applied successively $j - i$ times to arrive at p_i. Relying solely on $f()$ to contact p_i will entail more steps.

This proves our theorem. □

Let us assume an asynchronous network consisting of N faulty but not Byzantine processes. We can never achieve an algorithm for consensus if $t \geq \lceil \frac{N}{2} \rceil$ processes crash. A detailed proof for this can be found in [11]. This has an important bearing in our work, and the full import of this condition will be realized soon.

Let us now consider a portion of the bidirectional ring as shown in figure 2.2. Nodes v, u, w and z are initially connected. Process u wants to leave the ring, but its right neighbor w has crashed in the midst of the *leave* protocol.

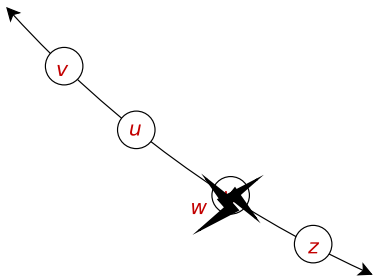


Figure 2.2: Crashed process in a bidirectional ring

This example illustrates the major problem faced by the algorithms listed in [3]. Since we are dealing with an asynchronous network, messages from all the correct processes arrive eventually. Also we

assume that crashed processes stay crashed and do not recover in future. So processes *v*, *u* and *w* can run a *decision* algorithm any time to determine whether they want to continue the protocol or not. If any process feels that, one of the other two has crashed, it votes against continuation and vice-versa. A correctly working process and one which is *not unduly suspicious* of the others, will always vote for continuation of the protocol. *Moreover, the algorithm can be run multiple times during the course of the protocol*. If one process has crashed and a tie of votes happens between the other two, discontinuation of the protocol is the preferred course of action. When a discontinuation of the protocol occurs, we assume that all nodes revert to the initial state.

But we still have the problem of *u* not being able to leave the network. In case a consensus is achieved among *v* and *u* that *w* has crashed, node *u* re-initiates the protocol but with *v* and *z* now. Process *z* is the next live process after process *w* and by theorem 2.2.1, *u* is guaranteed to reach it. Our treatment of crashes is not fair to slow, unresponsive processes as they also get classed with crashed processes. Assume in the above case that node *w* had not crashed but was merely slow. But by the time it recovers, the topology of the ring has changed and the situation is illustrated in figure 2.3.

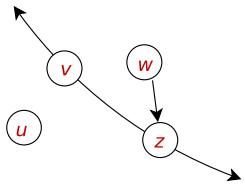


Figure 2.3: Nodes can be left out in a bidirectional ring

There is no proper solution for this. The only option then is to run a *repair* protocol initiated by process *z*, when it finds that two processes are considering it as its right neighbor. In this case, process *v* will have to change its right link to *w* eventually.

The above method is only 1-crash tolerant. But it is likely that more than one process might fail. So if we desire a more robust algorithm, which is say *t*-crash tolerant, then we do not have any other option but to start off with more nodes initially. In case of unidirectional rings, this is the only possible option. In subsequent sections we will further develop this methodology and supply fault tolerant versions of the algorithms listed in [3].

2.3 Joins/Leaves in Rings

The results in this section apply equally to both the unidirectional and the bidirectional rings. Consider the join protocol as illustrated in figure 1.3. It is obvious that there cannot be a 1-crash fault-tolerant *join* protocol in this case. Node *u* cannot determine asynchronously whether *v* has crashed or is slow. The same is the case for process *v*. To make it 1-crash tolerant we need another process. In our *decision* algorithm we use *u*, *v* and *w*. We assume that a 0 vote means discontinuation of the protocol and a 1 vote favors continuation. The *decision* algorithm to arrive at a 1-crash tolerant protocol is simple now. The *decision* algorithm for leaves remains the same. No special change is required.

begin
wait *for* $\lceil \frac{N}{2} \rceil$ **votes**
if *majority() = 0* $\rightarrow$ **stop();**
  $\triangleright$ *majority() = 1* $\rightarrow$ **continue();**
  $\triangleright$ **stop() fi**;
end

If we want to make the protocol more than 1-crash tolerant, we need to add more processes. Let us assume that we desire a t-crash tolerant algorithm. The situation is depicted in figure 2.4

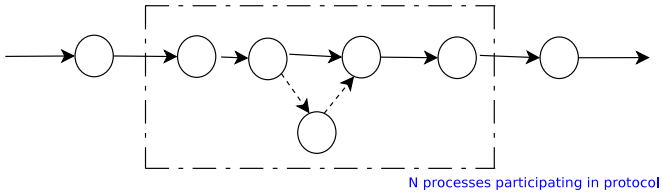


Figure 2.4: A t-crash tolerant protocol

So if we start with a set of N processes participating in our join protocol, the number of faults that can be tolerated, t, will always have to be less than $\lceil \frac{N}{2} \rceil$. Faults equal to or more than this limit will make our protocol fail. In figure 2.4, assume that the 2 nodes at the extremities of the inside box have crashed. There are a total of 5 processes inside the box and barring the 3 involved in the join, the other two have crashed. The join protocol can still be continued as the 3 processes all vote in favor of continuation. This is the correct decision if we assume that nodes cannot detect whether their neighbors have crashed. If we assume otherwise, then at least 2 of the participating processes in the join, will notice that their neighbors have crashed and would vote against the protocol. The protocol would be stopped and again, this is the correct decision since the 3 nodes (which are alive) are totally disconnected from the main ring. Thus we have managed to make the our join protocol 2-crash tolerant in this case.

Theorem 2.3.1. *Our algorithm guarantees a limited progress property. A non-crashing process, desirous to complete either the join or leave protocol (depending on the case), will eventually be able to do so. For a set of N processes, the progress property holds as long as there are no more than* $t < \lceil \frac{N}{2} \rceil$ *failures.*

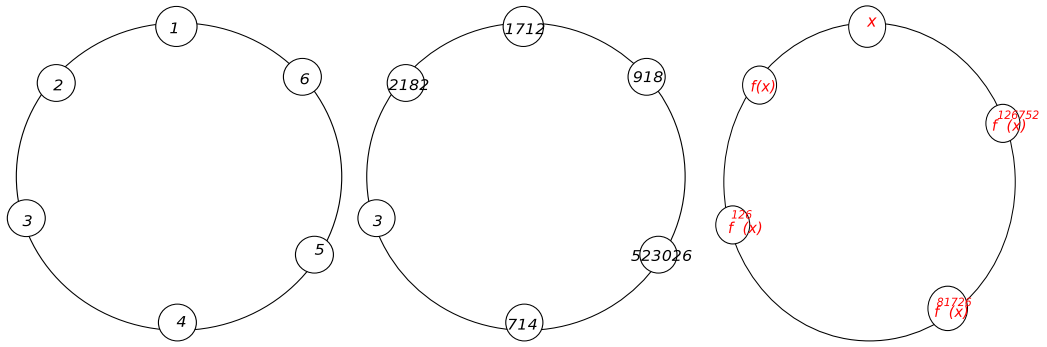
Proof. We assume that the process is denoted as u, and without any loss of generality we further assume that the protocol in question is the *leave* protocol. The initial situation would be similar to the case depicted in figure 1.5(a). The left and right neighbors of u are denoted by v and w. Assume that w crashes in-midst of the leave protocol, and the protocol consequently is voted as failed (processes not casting their votes are denoted as crashed; alternatively we can use out of band data to corroborate this). u will now re-initiate the protocol, but with a different set of processes, v and $f(w)$. If again a failure happens in the (new) right neighbor, the next iteration of the protocol will involve v, u and $f(f(w))$. If both v and w crash in the first phase, then the next iteration uses $f^{-1}(v)$, u and $f(w)$ and so on. We can continue this process up to t iterations, after which it is impossible to get consensus in the given asynchronous network [11]. □

2.4 P2P Networks

In many ways, P2P networks provide the best platform to test for robust, scalable, fault-tolerant algorithms. The wide popularity of such networks translates to a highly dynamic environment with many concurrent processes leaving and joining at any given time. Moreover, crashes do happen all too frequently. We seek to extend our scheme of using successor and predecessor functions to a P2P network, like the *wrap-around* ring based `Chord`.

Directly applying our scheme to P2P networks won't work for reasons given below.

1. *Usage of simple functions* This problem is illustrated in figure 2.5(a). If we use a simple function for $f()$, like $f(x) = x+1$ or say, $f(x) = ax$ (where a is some large number), then we are bound to



(a) Simple pre-order of id's (b) Unordered set of id's (c) "Pseudo" Sparse P2P network

Figure 2.5: Problems with $f()/g()$ in P2P networks

face this problem. In P2P networks, all the keys and process id's reside in a large address space, typically $[0, 2^{160} - 1]$. The number of processes in the network at any given moment is a very tiny fraction of this. If we assume even a few billion live processes in our network, then it translates approximately to 2^{30}. In P2P networks, like `Chord`, a node is responsible for storing all the keys falling in the range of its id and its predecessor's id[3]. So if we use a regular function, and the number of keys is far larger than the number of processes, the last node in the network will in effect store *almost* all the keys. Our P2P network will now resemble a traditional client server architecture, with the last node acting as server and all the other nodes as clients.

2. *Ordering of processes* Let us assume that we employ some symmetric encryption function as our function $f()$. Encryption functions have some nice properties which were well utilized in ring networks. Applying an encryption function would lead to the situation depicted in figure 2.5(b). We have id's randomly scattered across with no definite ordering, and implementing any kind of routing for such a network is nearly impossible. The central feature of most traditional P2P systems is the ordering of nodes around the identifier circle. But with an encryption function the nodes become totally disordered, and no traditional P2P system can be implemented on top of the given network.

3. *"Pseudo Sparsity" in the ring*[4] P2P systems witness a lot of traffic with a high amount of churn in the system. There have been attempts to measure and analyze this behavior, most importantly in [9]. Over a period of four days, they were able to record 1.2 million active peers in total. This study was done in 2001. It is not unreasonable to assume that currently one can record a few million active peers using some popular P2P networks (like Bittorrent, Napster etc). Moreover, [9] recorded that around 80% of the peers are active for a short time only before disconnecting. This is a very important observation. As mentioned earlier, two nodes in our scheme just have to compute $f()$ a finite number of times, before they can contact each other. Going by the measurements of traffic in P2P networks, it is extremely likely that the immediate neighbor for some node would lie a million iterations away. If we assume a simple hash function like SHA-1 (160 bits) as our $f()$, one iteration will take around 0.067 seconds[5]. So just to compute the node id for the immediate neighbor, some half a million iterations away, will take close to an hour then. Note that this is just

[3] meaning the key falls in the interval between its own id and its successor's id.

[4] By "pseudo sparsity" we mean that while the network might be otherwise dense, the computational cost for calculating the successor and predecessor functions is very high. This is different from the normal definition of sparsity and hence the term "pseudo sparsity".

[5] *http://www.hashemall.com*

to compute the node id, the resolving cost is extra. An interesting observation is that this problem is not endemic to P2P networks alone. If we apply our scheme to large rings with a high amount of churn, then this problem will recur there also.

To apply our scheme of successor and predecessor functions, we have to surmount the above difficulties. We propose two modifications of our scheme which will make our approach feasible for application to P2P networks.

1. *Using monotonic functions* For *f()* we propose to use monotonic functions. We can use either a monotonically increasing or decreasing function. A monotonically increasing function has the following property. If x_i and x_j belong to the domain of f and $x_i < x_j$, then $f(x_i) < f(x_j)$. Similar is the case for a monotonically decreasing function. Many simple functions like $f(x) = x+1$ will qualify as monotonic. As explained before, these cannot be used in our scheme. We do not supply an analytic expression for a suitable $f()$ in this work, but it should be possible to easily design a monotonic, non-simple function to fit our needs. In this work, we take an alternative approach and pre-compute the list of possible node id's. For example, we run the MD5 hash algorithm successively on some sample input key a few million times. This list is then sorted in ascending order and the resulting list can be widely replicated in different locations. Alternatively this "master" list can be broken down and stored at various locations (in case server space is at a premium). Any process desiring to join our network can pick up the next available id from the list and proceed to join the network. The id is then removed from the list of available id's. A node can also use a $contact()$ function to get the process id of last joined process, and run $f()$ on it to obtain its own id. If we have such a precomputed, sorted list of id's, then function $f()$ simply involves moving one place down the list, and its inverse, $g()$, is moving one place up. The size of this list will be quite small (in terms of the capacity of modern disks), and we can even expect nodes to carry large chunks of it. We wish to emphasize once again that in practice, using a suitable analytic expression for *f()* is more preferable compared to our method of pre-computation. But theoretically it does not make much of a difference.

2. *"Reincarnation" of process id's* We apply a scheme of "reincarnating", i.e. used process id's are again re-inserted into the system. Any process voluntarily leaving the network, will add its id to some publicly available list. A node desiring to join, can obtain this "used" id and use that for joining. There is no centralization here. Lists of "re-usable" id's can be stored in different places on the Net, corresponding to different regions of the network. Also an id cannot simultaneously exist in both the "free" list and the "reusable" ones. Crashed nodes should not be able to store their id on the "reusable" list immediately, and consequently their id will take some time to appear. If the number of crashed processes is small, this would not pose much of a problem. The whole scheme of "reincarnating" process id's, is to make the network denser. We want every node to ideally compute $f()$ only a few times before it reaches the next (live) neighbor's id.

Our approach for processes joining the P2P network is directly *opposite* to the normal convention. For example, in a system like `Chord` a process computes the hash of its physical (IP) address, and that becomes the id for that node in the network. We do not give this freedom to the processes in our network. A node joining the network has the following two options (in decreasing order of preference). It can obtain at random an already used id from the list of "reusable id's", or it can obtain the next available id from the list of "free id's" and proceed to join the ring.

Theorem 2.4.1. *Assume N processes distributed over a circular ring. If $N/2$ processes crash or leave voluntarily, and $\frac{2}{3}$-rd of the exited id's are "recovered", both at random, then on average, every process can contact its nearest neighbor on the ring in a small number of finite steps.*

Proof. If $\frac{2}{3}$-rd of all process id's get reincarnated, then the ring is always $\frac{5}{6}$-th full. The distance between two nodes is measured by the number of iterations of $f()$ required to reach the next live node. In case of a fully populated ring, the separation would be 1, since we need only one iteration of $f()$ to get the id of the next process in the ring. If the ring is $\frac{5}{6}$-th full, then the number of iterations is < 2, on *average*. Assume that this is not the case. In case the average distance is ≥ 2, then there can be a maximum of only $N/2$ processes in the ring. This disproves our starting assumption, and hence the average number of iterations to reach the neighbor is < 2. The maximum separation between two live nodes (worst case) would be $N/6$. But in case, crashes and re-incarnation of process id's happens totally at random, it is very unlikely that such a event would occur[6] (probability is $1/\binom{N}{\frac{5}{6}N}$). □

By theorem 2.4.1 it is obvious that, even if the number of processes in the system is very large, a node can still contact its nearest neighbor within a small number of finite steps. We can also minimize the message transfer incurred in this operation by making the process generate the id's of, say the next ten processes down the line, and then looking them up in a single message by bundling them together. The cost for such a combined look-up would be cheaper than looking them up one by one.

Corollary 2.4.2. *With high probability a node can find its immediate successor in an N-node network within a small number of finite steps $O(k)$ where $k < \log N$.*

Proof. For $N > 100$, $\log N$ will exceed 2. But if the conditions of theorem 2.4.1 are satisfied, then the average number of iterations to reach the next neighbor of the process (which will be the immediate successor) on the ring will be < 2. This completes the proof. □

Using the concept of successor and predecessor functions along with *finger* tables[7] normally used in `Chord`, one can obtain a highly fault-tolerant and dynamic P2P system. Also since we use a pre-sorted hash list to generate $f()$, the P2P system will be well-balanced. Note that the id's which are not "reincarnated" (but instead are pulled from the list of "free id's") will only be a single hop away from the next neighbor, since the next value of $f()$ is used for them. Moreover, our system provides limited auditing functionality. By noting the number of used id's, one can get a rough idea of the number of nodes in the system. This can be fine tuned further if one takes into account the number of id's waiting to get reused.

2.4.1 Joins and Leaves in P2P Systems

For the P2P system to be highly efficient, it is important that the processes in the ring have an accurate idea of the nodes comprising their predecessors and successors. `Chord` uses a stabilization protocol, *stab()*, which runs periodically in the background. This will update the successor list of a given process periodically [10, 4]. Before we furnish updated versions of *stabilize()*, *join()* and *notify()*, we need to highlight the fault tolerant aspect of our network.

Fault Tolerance

The key to building the *finger* table in `Chord` rests on a function called *closest_preceding_node*. The pseudo code for this is listed in [10], and we replicate the same below[8].

[6] *A more rigorous bound can be obtained for all the cases mentioned. But for the purposes of our design, these rough bounds suffice at present.*

[7] A finger table is used to increase speed of node lookups. If a finger table has *m* entries, then essentially the node will cache the locations of all its successors in increasing powers of 2 till it reaches its 2^m-th successor in the ring. Row *i* then corresponds to the 2^i-th successor in the ring.

[8] We abandon Gouda's abstract protocol definition here and revert to a somewhat Pascal-like notation

```
% search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger(i) ∈ (n,id))
      return finger(i);
  return n;
```

Let us for a moment assume that the nearest finger returned has crashed. There is no clear way to handle this in normal `Chord`. The query will be blocked, and we can hope to resume it once the network is stabilized again. But in our updated system, we can "jump" over the crashed node. Assume that the average separation between two live nodes is x (a small, finite number). Now $f()$ applied x times over the id of the crashed node will, with high probability, return the id of the next live node in the ring. Moreover, we can vastly improve upon this if the network is suitably dense (if the conditions of theorem 2.4.1 are satisfied). To find a successor for a given process id n, we just run $f()$ a finite number of times on n, and then do a single look-up on the process id's thus obtained to find the first available/live successor.

Lemma 2.4.3. *In an N-node network, the number of nodes to be contacted to find the successor of a process id is* $O(\log N)$. *But if the network is dense, it can be found in* $O(k)$.

Proof. The proof for the normal case of $O(\log N)$ is listed in [10]. If the network is dense, finding the successor is a simple matter of looking up the first live process from the set of process id's obtained by running $f()$ on the id a finite number of times. □

Lemma 2.4.4. *Given a key k, finding the node responsible for it will involve* $O(\log N)$ *steps if the network is sparse, and* $O(k)$ *(where* $k < \log N$*) if the network is dense.*

Proof. For a sparse network the proof is listed in [10]. For a dense network, the client can pre-compute the approximate node responsible for the key (by referring to tables of pre-computed values of $f()$ for example) and then use $f()$ to start the search from this node. The number of steps thus taken will be less than $\log N$ since we are already starting close to the responsible node, if not at the actual node itself. □

Dynamic Operations

In our ideal system, the density in the network is high, i.e. a small number of finite iterations of $f()$ will lead us to the successor of any node[9]. This fact is important because based on this we can implement a rich set of API's for our system. A function like *query_predecessor(n, trail)* will return the first live node, which is not trailing the node n by more than $trail$ number of iterations. We use this to modify the standard *join()* of `Chord`.

```
% ń uses an existing node n to join the system
n.join(ń)
  predecessor = nil;
  s = ń.find_successor(n);
  build_fingers(s);
  successor = s;
  m = query_predecessor(ń, trail);
  if m != nil
    predecessor = m;
```

[9] In case the network is static and with no churn, there is no problem to begin with. On the other hand, if the network is such that nodes keep on leaving/crashing in high numbers, without a corresponding proportion of joins, then our model will fail. In the latter case the system reverts to a `Chord`-like behavior to stabilize the network.

We can also create a function *query_live_nodes(a,b)* which will return any live processes between *a* and *b*, provided the two parameters are not too far apart. Such a function can be used fruitfully in redesigning the *stabilize()* function of `Chord`.

```
% periodically verify n's immediate successor,
% and tell the successor about n
n.stabilize()
  x = query_live_nodes(successor,n);
  if x != nil
    successor = x;
  successor.notify(n);
```

Note that the functions are to be used when the range between the two parameters is small. Else the time taken to return will be enormous. We can alternatively return **nil** in such cases. Also we do *not* do away with the *finger* tables used in `Chord`. Our design simply enhances it and makes the system more fault-tolerant and fast. This brings us to the following lemma.

Lemma 2.4.5. *In a sparse network or a network where the rate of leaves and crashes far outweighs that of joins, our system reduces to that of* `Chord`*.*

Proof. All our functions return **nil** in case the ranges are large. Moreover, in a sparse network, a small number of iterations of *f()* will return no live processes. This forces the system to rely solely on the traditional *finger* tables used in `Chord`. This proves our lemma. □

Lemma 2.4.6. *If normal system operations are measured over a suitably large time interval, the rate of leaves and crashes cannot exceed the rate of joins to the system.*

Proof. Assume that the rate of leaves and crashes exceed that of joins. In that case, the membership of the system is declining and soon it will reduce to nil. This is against our assumption of a system which is constantly online. □

We can know whether the rate of leaves outweighs that of joins, by just checking the activity in the tables of "reusable" id's. If the tables keep growing faster over time, then it means a high rate of leaves. This condition can arise in modern day systems due to major WAN disturbances like, for example, onset of a worm such as `Slammer`. But generally these conditions are temporary and normal order soon gets established.

2.5 Conclusion

Using successor and predecessor functions, we have demonstrated how to create highly responsive and fault tolerant, rings and P2P networks. A good feature of this method is that we can always fall back on the more traditional means of routing and communication, should conditions not be ideal for applying these functions. Our worst-case performance is the same as that of `Chord`, while in normal situations it would far exceed `Chord`. In the coming chapter we seek to improve upon this method and come up with even more robust and faster networks. While the basic principles remain the same, we utilize some more assumptions.

3 A Special Case

There have been many studies conducted on P2P systems which mainly test the homogeneity (or lack thereof) of the various peers and measure the connectivity, latency and sharing ratios between them. We combine the ideas listed in the earlier chapter along with the results published in [9]. One of the important findings of [9] has been that the "best"[1] 20% of peers in two popular P2P networks had the best up-times also. This very clearly proves that *not* all the peers participating in the P2P system are homogeneous. We seek to leverage this vital information and come up with a new P2P system, which takes into account this issue. We assume the very basic of information to start with. The P2P system designer need to have only a rough idea of the size of the user pool and the percentage of the "best" peers in the system. *Our goal is to come up with a system which takes into account this inherent non-homogeneity and yet is stable, fault-tolerant and very fast.*

3.1 Related Work

Our design can be classified as a OneHop peer-to-peer system. OneHop peer-to-peer systems have been the subject of intense research in recent times. There have been studies which explore its scalability and robustness especially under heavy churn [2, 5]. The current notion is that OneHop is preferable to `Chord` in cases of up to 3000 nodes under heavy churn. OneHop or O(1) networks are more suitable compared to `Chord` when the network is relatively stable. In cases of networks with many ephemeral nodes[2], `Chord` and other similar $O(\log N)$ networks are the preferred choice [7]. Our approach resembles the one adopted in [5]. Conceptually our design has been realized by exploiting the ideas listed in the previous chapter and we give proof sketches, which show our network providing constant lookup time, close to $O(1)$ even under heavy churn. We hope that our design can ultimately help to bridge the performance gap between current $O(1)$ and $O(\log N)$ systems.

3.2 Basic System Design

Our system consists of two types of nodes, as shown in figure 3.1. *Primary nodes* represent peers which have high speed connectivity and are up most of the time. They are also the ones that contribute most of the shared data. In [9] the percentages of these peers is shown to be around 20%, which means for every primary node, there are five other nodes which are poorly equipped. Membership of the primary nodes is not fixed of course. Rather, at any time we can only expect a certain percentage of peers to fit in the profile of primary nodes.

Secondary nodes shown in the figure comprise those processes which join the P2P system from time to time. They have little uptime and comparatively poor connectivity. These represent the bulk of the activity. Normally they do not contribute much of the data and are mostly interested in downloading

[1] Those stable with high speed connectivity. Normally they have more data to share also.

[2] nodes which have a short lifespan in the system

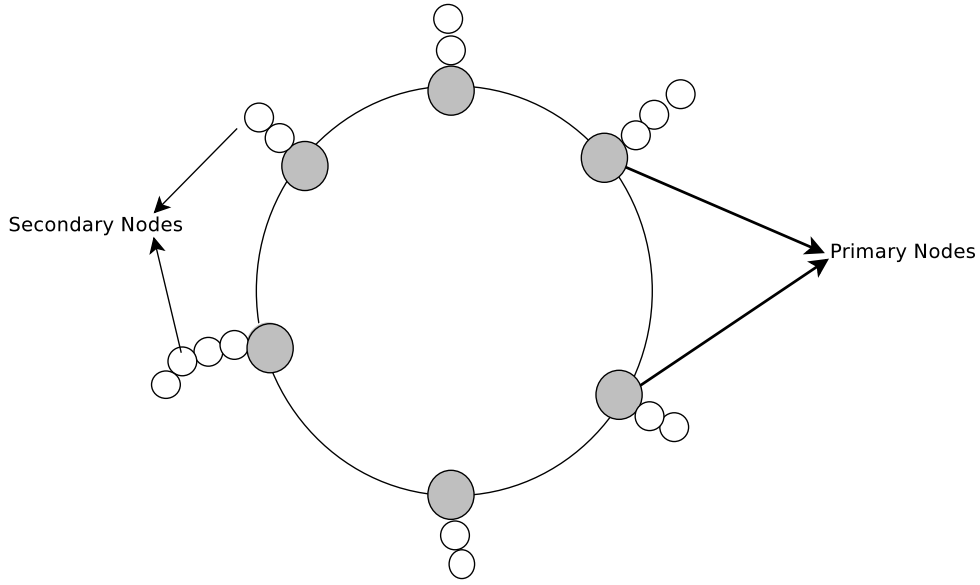


Figure 3.1: Proposed P2P system

only. We assume that nodes which join the system are aware of the identity they will assume and the distinction of roles.

Here we come to the first major question. How do we *apriori* determine which processes qualify as "primary" and which as "secondary"? There is no satisfactory way of doing so. One way is to measure the upstream and downstream bandwidths of the participating processes (expecting the participating process to reliably provide this information won't work, as [9] has shown that most peers falsely report this information). Nodes with hi-speed connectivity qualify for being a primary node, else they are relegated to secondary role. This is obviously not a fool-proof method, but it will suffice most of the time. The authors in [9] list methods to develop tools which can accomplish the bandwidth measurement economically.

3.2.1 Structure of the System

We assume that a suitable symmetric encryption algorithm, $K_p()$, has been used to generate the process id's in the core ring. Starting from a random value (start id), a set of id's is generated by repeatedly applying K_p to it. We are assuming a wrap-around ring here, so after a finite number of iterations it returns back to the start id. Primary nodes are given any of the id's from this set, depending on availability. Primary nodes leaving the system will make available their id for reuse. This scheme is a bit flexible. If we desire to increase the number of primary nodes, we make available a larger number of id's in the system, and we can also cut down id's from the list of reusable id's to limit the number of primary nodes. A node which joins the core ring is made aware of its role as a primary node. Generally we view primary nodes as more dependable than secondary ones.

To generate the secondary nodes we use another symmetric encryption algorithm denoted by $K_s()$. By using a totally random election procedure, we *attach* the secondary node to any of the primary nodes. Since allocation is done totally at random, we can expect every primary node to have an almost equal number of secondary nodes attached to it. Going by the results in [9], for every primary node there will be upto five secondary nodes. This would seem small and a drastic simplification. In our design though, we assume that the number of secondary nodes assigned to a primary node is quite large and with no significant limitations.

Let us assume that the keys and process id's reside in a 160-bit address space, i.e. $[0, 2^{160} - 1]$. Initially this space is divided equally among all primary nodes. So if there are N primary nodes, each node is responsible for $2^{160}/N$ keys. Primary nodes which are voluntarily leaving the network relinquish their share of keys to their successor in the ring. The new node replacing the vacancy will take its share of keys from its immediate successor. From figure 3.1 it is clear that a primary node "manages" the secondary nodes attached to it. This entails a simple set of operations. A secondary node gets allotted to some primary node by a random process. It collects its id from the primary node (either a used one or the primary can generate a new one, depending on the case). Also a certain share of keys is assigned to the new secondary node. Each time a secondary node joins, the primary will assign x% of the keys it has to the new node. The value of x can be arbitrary (a good choice would require some more data about the traffic, etc), but the important thing here is that the value of x is known to all nodes in the network. A secondary node will transfer its keys to the primary when it is voluntarily leaving the network. Thus at any point, a primary node has a good idea of the number of secondary nodes it is responsible for. It can also advertise this figure openly to the other primary nodes too. Since the allocation of secondary nodes is done totally at random, with a high probability all nodes have about the same number of active secondary nodes. This has important consequences, which will be detailed in later sections.

3.2.2 Dynamic Operations

We have covered the question of joins and leaves of the various components of the system in the earlier section. We now turn our attention to crashes. A primary node, by assumption, is not prone to frequent crashes. Unfortunately it is impossible to provide a 100% guarantee. The moment a crash is detected (presumably its neighbors on the ring or its secondary nodes find the crashed node unresponsive over a long period), other primary nodes are informed (for instance by gossiping), and the vacancy in the core ring is advertised. The successor of the crashed node assumes responsibility for the keys covered earlier by the crashed node. A leave by a primary node is more graceful. It just transfers its share of keys and information about the number of secondary nodes to its successor. All messages from the secondary nodes to the departing primary node are now diverted to this successor. We always strive to aggressively fill in any vacancies caused by crashes or departures in the core ring. Crashes of secondary nodes are less of a concern. The primary node responsible for that section comes to know sooner or later, and it will take care over the keys residing in the earlier crashed node, and also add the id of the secondary node in its list of reusable id's. Leaves among secondary nodes proceed in a similar manner, except that keys are handed over to the concerned primary node and the primary node again adds the id of the departing node to its list of re-usable id's. Note that there is no re-assignment of keys in the event of a crash/leave of a secondary node. The concerned primary node will simply take over the management of the keys which used to reside in the departed node. Our logic behind this is that in a highly dynamic system, vacancies are filled as fast as they are generated.

The information available to each type of node is very limited. Both types of nodes know to which class they belong, whether primary or secondary, the values of keys used (K_p/K_s), and system related information like x and the size of the primary ring. Secondary nodes know their id, the range of the id's responsible by its primary node, and possibly the number of iterations of $K()$ they are from the primary node. Primary nodes know a bit more. They know their own id, the number of secondary nodes they possess, and the full range of id's they are responsible for. *We do not propose to use finger tables, routing maps, neighbor tables and such in our construction.* There is no elaborate background protocol for stabilization of the network. The only information a node can possess has been listed above.

3.3 Analysis

We now proceed to demonstrate how we can achieve fast routing using our system. The following theorems are applicable only to a fault-free environment. The situation with faulty environments will be dealt with later.

Theorem 3.3.1. *Given a key k, any node can always determine the identity of the primary process responsible for the key.*

Proof. By our construction, any node knows the identity of its primary node and the range of keys that fall under the responsibility of its primary node. Furthermore, the number of primary nodes in the system is a design parameter and known to all participating processes. Given this information, it is easy to compute the number of hops away from the current primary process, the new key would be residing[3]. The id of the primary node can then be computed by applying $K_p()$ or $K_p^{-1}()$ on the id of the node's primary process, the desired number of times. □

Theorem 3.3.2. *Assume that a secondary process is i hops or i-iterations away from its primary node in a chain of j nodes. It can resolve all keys belonging to other primary nodes, in $O(1)$ and keys residing in other secondary nodes in $O(1)$, but with a probability of i/j.*

Proof. The allocation of secondary nodes is done totally at random. So every primary process has almost the same number of secondary nodes at any given time. Thus a given secondary node can assume that with high probability other primary nodes have approximately the same number of secondary nodes, namely i. By theorem 3.3.2, a secondary node can point out the primary process responsible for that particular key. This will be done in $O(1)$. Furthermore with high probability we can assume that $i \times x\%$ of the id's residing in the given primary have been shifted to its secondary nodes[4]. Now with a probability of i/j the given id resides in the first i nodes, and our secondary node will be able to resolve it (by applying $K_s()$ that many number of times on the id of the primary node). If the given key does not reside in the first i nodes, then the secondary node will assume that the primary is still in control of that key, which is erroneous. □

Corollary 3.3.3. *If a secondary node knows the total number of secondary nodes in its own chain, then it can resolve most queries in $O(1)$.*

Proof. The crucial fact here is that almost all primary processes in the ring have approximately the same number of secondary processes at any given time. By theorem 3.3.2 we can calculate the id of the primary process responsible for that key, and then work out if the key resides in the any of the secondary nodes (note that secondary nodes keep taking $x\%$ of the keys from the primary always), or if the primary is still holding the desired key. This completes the proof. □

Corollary 3.3.4. *A primary process can resolve most queries in $O(1)$ with high probability.*

Proof. A primary process will always know the number of secondary nodes it has and by our assumption the number of secondary nodes every other (primary) process has. By theorem 3.3.2 it can now resolve most queries in $O(1)$. □

Corollary 3.3.5. *A random secondary process can successfully resolve any given query in $O(1)$ with a probability just over* 0.5.

[3] We can do a simple computation for this. If the number of primary processes is a power of 2, say 2^k and the total address space is again 2^j, then every primary node is responsible for 2^{j-k} entries. Right shifting the given key $(j-k)$ times till we get 0 will reveal the number of the primary node responsible for the key

[4] Without loss of generality assume that the id's are leeched from the starting range of id's the primary is responsible for.

Proof. By theorem 3.3.2, in a chain of j-secondary nodes, the i-th secondary node will resolve all queries successfully with a probability of i/j. Thus on an average, a secondary node can successfully resolve a given query with probability $\frac{\sum_{i=1}^{j} \frac{i}{j}}{j} = \frac{\frac{1}{2}j(j+1)}{j^2} = \frac{1}{2}.\frac{j+1}{j}$. □

Now we turn our attention to faulty environments. A primary node can crash with a low probability, but no such assumption can be made about the secondary nodes. The main point of interest is that the keys residing in crashed or departed secondary nodes get transferred to the primary in finite time. Also the successor of a crashed or departed primary node will take over the responsibility of its neighbor as long as the vacancy remains unfilled. Finally since the secondary node prior to joining has to pick up an id from the concerned primary node, we can safely assume that the primary node has a good idea about the number of active secondary nodes that it is responsible for.

Theorem 3.3.6. *In our given P2P system, most queries are resolved in $O(k)$ where k is a small, finite number.*

Proof. A secondary node can find the query to the concerned primary node in $O(1)$. The primary node knows the number of secondary nodes it is responsible for, and based on this can decide whether the given id resides in one of its secondary nodes or with itself. Finally, it routes the query again to the concerned secondary node or to itself in the next step. This completes the proof. □

The bulk of the theorems from the earlier chapter will also apply here, especially relating to joins and leaves. A process is guaranteed a successful join as long as the primary node it is attached to does not crash during the join process. In case a crash in the primary occurs, a node can always apply to join another primary (using $K_p()$ over the initial primary node id will give the address of the next id on the chain), or it can wait a finite time for the primary node to return. We are guaranteeing a *progress* property here. The same is the case with leaves. The fault tolerant nature of the system is very important. Since we directly home in to the concerned nodes, even if the bulk of the nodes on the way have crashed or are crashing, it will not affect our progress. This is a trait missing in other comparable systems. The more the number of steps in routing, the less fault-tolerant the system becomes.

3.4 Conclusion

Using the concepts from the earlier chapter and some more assumptions, we have shown a special case of P2P networks which guarantees fast routing. The remarkable fact of our design is the complete absence of neighbor maps, finger tables and such. There is very little background stabilization of the network. In fact, we have exploited the inherent non-homogeneity of modern day P2P systems to our advantage. We can speed up our network if extra functionality like routing tables or neighbor maps and such is added. But the base design is sufficient and robust enough. So adding more complexity to the network should not be required. To lessen the computational load on the nodes, we can distribute pre-computed tables of $K_p()$ and $K_s()$. This should help in reducing the computational workload in all the nodes.

We hope to conduct future experiments to test the performance of the networks listed in this and the previous chapter. There are numerous parameters of this system which need to be empirically determined, prominent among these is the average distance between two nodes when the system is under heavy churn. Also it is very important to fix the limits for the percentages of crashed nodes (as opposed to those leaving gracefully). Crashed nodes would take some time to get reincarnated. Nodes departing voluntarily get their id added to the “available” list more or less instantaneously but it would take some time for nodes to realize that one of its neighbors has crashed. If we imagine a network under heavy churn and crashes far outweighing voluntary leaves, then there is a possibility of the system performance degrading. Thus it is important to determine the extent of crashes a system can gracefully handle. We

need to experimentally determine the limits that the network can scale up to, and the speed ups realized, and compare it to the existing simulations done on `Chord` and $O(1)$ networks. We plan to mainly use the popular network simulator `ns-2` for this work. We hope to simulate both kinds of networks, $O(1)$ and $O(\log N)$, but built using our ideas. Currently there are simulators existing for `Chord`. But right now, we are unable to say definitely, whether any of the existing simulators can be modified to suit our design. Furthermore, we hope to construct a rich set of API's on the system. We expect that interesting functionalities like caching and replication can be built in a robust and scalable manner with our construction.

4
A Partial Geometric Solution

In this chapter we seek to provide a *partial* geometric solution to the problem of designing fault tolerant rings and networks. The solution is not complete as there are many open questions, but the approach looks promising and has therefore been added to this work. Hopefully, future research would answer some of these questions. The whole idea in this chapter and the previous two chapters have been to come up with designs of networks with very fast routing. This automatically leads to more fault tolerance, since with fewer nodes hindering it, two active processes will always be able to communicate. We continue in much the same vein here. We begin by constructing a network in which two nodes have a multitude of paths to communicate with each other. Just by knowing their id and the destination id, we can guarantee communication.

4.1 Overview

Assume a *hyper-ring* system as shown in figure 4.1. Basically by an *hyper-ring* we mean a system of concentric rings which are interconnected with each other. A single ring can hold a maximum of r_s nodes, and there can be a total of R rings in the system. Both these parameters are system dependent and should be defined appropriately by the system designer.

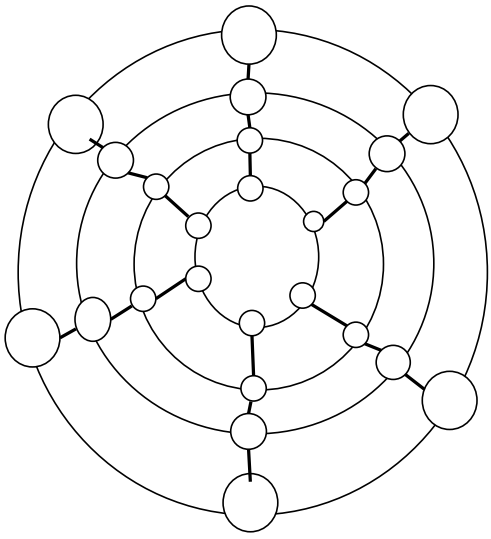


Figure 4.1: A hyper-ring system

A node in the interior ring can have a maximum of four neighbors. Nodes which reside on the ring(s) on the extremes of the network can only have three immediate neighbors. We use a N size namespace ($= R \times r_s$), with id's represented in base b. The most significant bit of the process id would represent the

ring it belongs to. Let us assume a single ring with slots for N nodes. So we can distribute nodes with id's a, ar, ar^2, ar^3 and so on till ar^{N-1}. Basically the node id's will form a geometric progression. Any id $\leq a$ will be mapped to the first node, those greater than a and $\leq ar$ to the second node and so on. This is only one possible arrangement, but we can always try others too[1]. Disused id's are recycled back into the system to make it denser. In case there are no id's left to be reused, then nodes prior to joining run a local algorithm at their end which will throw back any (free and unused) id at random. Using this they proceed to join the network. In case the id has already been taken, we revert to running the algorithm again. Leaves are also handled in a similar fashion. It hands over the keys it is responsible for to its successor, and puts up its id for re-use. Neighbors of the departed node will then break the existing link and instead connect to its successor to keep the network topology intact. In this network we assume that all peers are reasonably homogeneous to each other and are liable to crash or depart with equal probability[2].

4.2 Analysis

How fast can such a network route? In case the network is dense and more or less adheres to the topology in figure 4.1, it is trivial to show that all routing will happen in $O(1)$. But the performance deteriorates as soon as sparsity sets in. This forces us to use neighbor maps in much the same manner as in `Tapestry`. Each node in the network can have a maximum of four neighbors. Assume that it keeps a neighbor map which shows the node responsible for particular positions in its own ring, and the rings above and below it. Thus the size of the map would be $3 \times r_s$.

Theorem 4.2.1. *If the given network is sparse, then normal time for routing to a given key is around $O(R/2)$, where R is the number of rings in the hyper-ring.*

Proof. We use the neighbor map each node has and navigate ring by ring. Assume that the nodes in a single ring can be numbered from 0, 1, 2, ..., $R-1$. From the given key we can determine the node which is responsible for it (since the ideal distribution pattern is known beforehand) and the ring on which the node belongs (this is from the most significant bit of the key). Assume that the key resides in node i in ring number j. We first navigate to ring j, and then proceed to find the node i (or the corresponding responsible node in case i does not exist) in that particular ring, using the neighbor maps already constructed. The average number of hops would then be

$$\frac{1}{R}\sum_{i=1}^{R}\left[\frac{1}{2}\sum_{j=1}^{R}[R_j - R_i + 1]\right]$$

This is equal to $R/2$ and completes our proof. □

The routing performance of our network does not compare favorably with the other networks discussed in earlier chapters. But if the size of neighbor maps is increased to encompass more rings, then the routing performance will increase correspondingly. This comes at a cost though. The bigger the size of the map, the larger the cost for background maintenance (in terms of messages, complexity of joins/leaves and computational overhead).

4.3 Problems

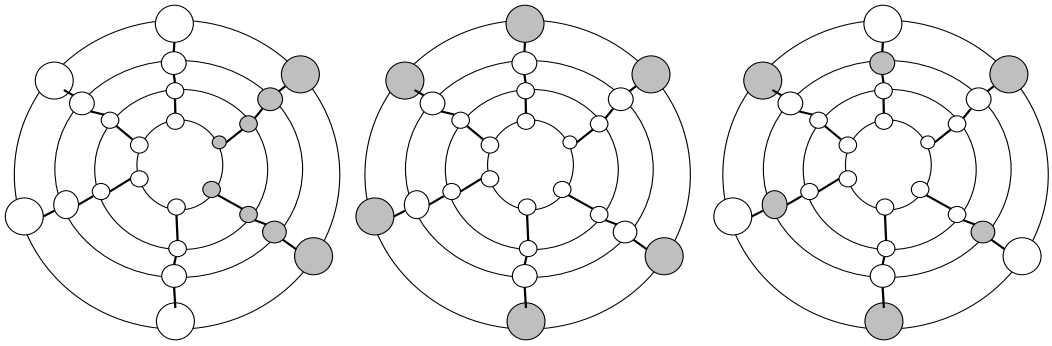
The network analyzed has a number of significant problems, which we detail below.

[1] This is in fact one of the open questions.

[2] This need not always be true, as detailed in the earlier chapter.

- *Apriori information* To get this idea working properly, we need to know beforehand an accurate figure for the membership expected. This is needed most importantly to determine the size of the rings. It is also important that the rate of joins and leaves closely match each other. If membership figures are over-reported, then we will end up with more rings than required, and a lesser figure would be equally disastrous for performance. The same is the case if the rate of joins and leaves are mis-matched. Note that none of the earlier networks discussed had this kind of limitations.

- *Sparsity* This could lead to serious performance problems. It is evident from theorem 4.2.1 how the performance suffers on account of sparsity setting into the network. This leads to the question on how best to fill up the network. Expecting the full strength of members to be present initially would be fallacious. There could be different schemes for filling the ring as shown in figure 4.3. These are just a small sample of possibilities, and do not represent the only possible way of filling the ring



(a) Radially filling (b) Filling it ring by ring (c) Alternate filling on rings

Figure 4.2: Sparsity issues in a hyper-ring

  There are problems in each of the schemes. In case of radial filling, one or two crashes on the spoke will disrupt the whole operation. It is not clear how we can cope with that. On the other hand, filling it ring by ring will make the initial rings responsible for keys to be mapped for the other nodes. This might lead to performance loss, and if by some chance the outer ring gets disrupted then performance would degrade rapidly.

- *P2P networks* How to map P2P networks to this ring is an interesting question. There has been some prior work done on mapping P2P systems to constructs like *cube connected cyclic* networks (by Pandurangan et al. [6]). Possibly the same could be applied here. The main problem with the ring is that performance degrades rapidly in case of sparsity. How to bootstrap such a network is again an open question.

4.4 Conclusion

We have demonstrated how to construct networks based on concentric rings. The problems in such an architecture have also been discussed. But we still maintain that it is a promising line of research. In case node id's are "recycled", the network would become dense and routing could happen in $O(k)$ (k is some small number). Also it is trivial to prove that a dense ring would provide excellent fault tolerance and connectivity options. If satisfactory solutions are provided to some of the questions raised earlier, then the ring could emerge as a viable architecture for many different types of networks.

Bibliography

[1] M.G. Gouda. *Elements of Network Protocol Design*. Wiley, New York, 1998.

[2] A. Gupta, B. Liskov, and R. Rodrigues. Efficient Routing for Peer-to-Peer Overlays. In *First Symposium on Networked Systems Design and Implementation NSDI*, 2004.

[3] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent Maintenance of Rings. *Distributed Computing*, 19(2):126–148, 2006.

[4] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proceedings of the 21st ACM Symposium on principles of Distributed Computing*, pages 233–242, 2002.

[5] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured Superpeers: leveraging heterogeneity to provide constant time lookup. In *IEEE Workshop on Internet Applications*, 2003.

[6] G. Pandurangan and S. Jagannathan. A Simple Churn Tolerant Structured Peer-to-Peer Scheme. *submitted*, 2008.

[7] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: Search methods. *Computer Networks*, 50:3845–3521, 2006.

[8] John Risson, Ken Robinson, and Tim Moors. Fault Tolerant Active Rings for Structured Peer-to-Peer Overlays. In *Proceedings of the 30th Annual IEEE Conference on Local Computer Networks*, pages 18–25, 2005.

[9] S. Saroiu, P.K. Gummadi, and S.D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2002.

[10] I. Stoica, R. Morris, D.-Nowell Liben, D. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[11] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.