Abstract

Computing nodes in modern distributed applications on large scale networks form an overlay network, a logical abstraction of the underlying physical network. Reliability of data delivery is an essential requirement for many of these applications. Traditional methods of ensuring guaranteed data delivery among overlay nodes, are built on the idea of duplication of routing tables at multiple overlay nodes, to handle data delivery even when overlay nodes or links fail. This does not guarantee data delivery as the new routing path to the destination node may also contain the failed node or link in the underlay path. We claim that reliable data delivery in overlay networks can be achieved only through underlay awareness at the overlay level. We propose that reliable data delivery can be achieved by using highly available overlays. An overlay network with reliability guarantees has to be self organizing, due to the large size, distributed nature and cost of redeployment. The major contribution of this paper is a chordal ring based self organizing overlay which is self healing upto two node/link failures. We also present asynchronous distributed algorithms that are executed by the nodes that join and leave the overlay, which ensure that the data delivery guarantees of the overlay are maintained. The algorithms are proved to be correct under distributed and concurrent executions, and the time, space and message complexities are $O(\text{pathlength})$, where pathlength is the average path length between overlay nodes, counting underlay nodes of degree greater than two. The scalability of the algorithm is demonstrated with a simulation study.

1. Introduction

The logical paths for communication of data between the nodes in an overlay network actually pass through many physical links and nodes in the physical network, which are collectively called the underlay of the overlay network. The overlay substrate serves the purpose of hiding the intricacies of the lower layers from the applications. Figure 1 illustrates an overlay network $G_o$ on an underlying physical network $G_u$.

Due to the large size, distributed nature, cost of redeployment and dynamic addition/deletion of overlay nodes, intervention of an external entity for maintaining the network properties is not feasible. Hence an overlay network with QoS guarantees has to be self organizing. Self organization of overlay networks for maintaining properties like QoS guarantees has not received deserving attention from the research community. In the recent years, underlay aware P2P networks [16] are being explored by researchers for im-
Figure 1. An overlay network

proving the QoS guarantees for applications. In P2P over-
lays data delivery is done through overlay routing paths,
which are decided by the routing algorithm. Reliability of
data delivery is an important requirement for many appli-
cations. Each node routes data according to a routing table
generated and stored at the node. Overlay nodes are suscep-
tible to node failures. Underlay nodes, or links, may also
fail and possibly result in failure of overlay links. To ensure
reliability of data delivery, the network should be self heal-
ing to such failures, and routing of data to the target node
should occur despite the failures.

The traditionally proposed solution to overlay node fault
tolerance is to replicate the routing information on another
overlay node, and let the backup node do the routing when
the overlay node fails. We state that this approach does not
ensure reliability and illustrate it using Figure 1.

It may be observed from Figure 1 that even if node D’s
routing table is replicated in node B, for reliable routing,
node C would fail to receive the data it was meant to get
from B in the event of D’s failure. The overlay link BC con-
tains node d in the underlay, hence any data from B cannot
reach C, if D fails.

Hence, an alternate overlay node can provide the re-
quired data routing only if it is guaranteed that it has a
routing path which is independent of the failed node in the
underlay. Underlay aware overlay topologies can help in
selecting suitable overlay nodes to replicate routing tables.
We have developed a strategy for self organizing underlay
aware overlays with data delivery guarantees based on the

The overlay has an irregular chordal ring topology. Event
Broker Networks (EBN) are a typical example of over-
lay networks, and we consider EBN to be a backdrop
application to apply our strategy. Hence, in this pa-
per, we use the terms broker node/network and overlay
node/network interchangeably.

The main contribution of this paper is an overlay network
that is (i) Self Healing in the face of upto two broker node
failures. This means data delivery is guaranteed even if any
two overlay nodes fail. (ii) Self Organizing - overlay nodes
add themselves and withdraw from the network without re-
deployment of the application or affecting the self healing
property of the overlay.

We present distributed asynchronous algorithms for
overlay node joining and leaving the network and for main-
tenance of the chordal ring overlay, to maintain the self
healing property of the overlay network. The algorithms
are efficient, O(pathlength) in terms of message, time and
space complexities, where pathlength is the average num-
er of nodes of degree more than two between two over-
lay nodes in the underlay, and proved to be correct under
sequential and concurrent executions. The organization of
this paper is as follows - in Section 2 we state the necessary
theoretical concepts for our design, outline the requirements
and illustrate the formulation of our solution. Section 3 de-
scribes our solution, the underlay aware chordal rings and
the self healing overlay based on them. Section 4 presents
the distributed algorithm for the chordal ring overlay forma-
tion and maintenance. The simulation results demonstrating
the scalability of the technique is illustrated in Section 6.
Section 7 presents the related work in the field and Section
8 summarizes our contributions and concludes the paper.

2. Theoretical Background

In this section, we describe reliability of data delivery
and its sufficient conditions. We show that these condi-
tions lead to a chordal ring topology, with high availability
[6]. We introduce our solution, which is an underlay aware
chordal ring built on highly available overlays [5]. Our tech-
nique is based on two major assumptions, relevant for P2P
overlays.

(i) The routing of data items among the overlay nodes
takes place according to routing tables decided by the rout-
ing algorithm. Every overlay node has a routing table which
tells the node to which overlay neighbour it should send a
particular data item. This assumption is equally valid for

(ii) Routing path sets are mostly hierarchical, i.e., they
are structured as trees rooted at the source, especially in
event based networks for publish subscribe systems. This is
done to avoid duplicate event(or data) delivery.

2.1. Reliability of data delivery

We define the reliability of data delivery in an overlay
network as the minimum number of underlay network com-
ponents (i.e., nodes or links) whose removal (failure) causes failure of data delivery from any source overlay node to any destination overlay node in the network. For the hierarchical network typical of EBNs, the reliability of data delivery is clearly 1, as the removal of the parent broker node in the routing tree rooted at the source, would result in failure of data delivery.

The reliability of data delivery would be 2, if the routing table entry for the parent node (p) of the node x, had been replicated in another (backup) overlay node b, and it was guaranteed that in the event of failure of p, the backup overlay node b would detect and do the needed routing, and that there are no nodes(underlay or overlay) common to the path from b to x and p to x. Such an overlay would guarantee data delivery in the face of single node (or link, as node disjoinedness implies edge disjoinedness) failures.

We can deduce from the above reasoning that if the routing table entry is backed up in k other overlay nodes, and it is guaranteed that the failure of any of these nodes is detected by other nodes, and each has a path to the destination node which is disjoint at the underlay level from any of the other (k-1) paths, then a reliability of data delivery of k is obtained in the overlay and data delivery in the event of k-1 failures in the underlay is guaranteed. In this paper, we focus on reliability of three (k=3).

To summarize, we formulate the necessary and sufficient conditions for an overlay with a reliability of three as follows:

R.1. Every routing table entry is backed up in two more overlay nodes. These nodes would route the required data if the overlay node fails.
R.2. At least three (or more) nodes/links in the underlay have to fail simultaneously to isolate the destination node from all these three overlay nodes.

These properties should always be maintained in the network. The alternate nodes should detect failure and start routing. An overlay network design to satisfy both the above conditions is derived in this section. The chordal ring structure, described next is sufficient for the first condition. The second condition is met using a highly available, underlay aware overlay to build the chordal ring. The remaining part of this section shows how we reach this solution.

2.2. Chordal ring overlay

For a single node, backing up different entries of its routing table in different nodes would result in larger message overhead, for failure detection. Hence it is better to store whole routing tables into two other nodes, and have the mechanism (messages) for failure detection of the node in these two nodes. Thus every node should have two “special” neighbours. A rudimentary approach to this could be to form groups of three nodes, and let the members of each group be the special neighbours of each other. But this would create the restriction that the total number of nodes be a multiple of three, which is not practical to maintain. Moreover, node deletions would be difficult to handle, and it does not contain the concept of an order, to determine which of the two neighbours should start routing if a node fails. A better strategy would be to organize the nodes as a logical ring, so that each node has two “special” neighbours, whose routing tables it stores, and who store its routing table. This structure can be maintained independent of the number of nodes. Moreover, deletions could be easily handled, as the neighbours have to link up, without affecting the rest of the overlay. A ring connotes an ordering on the neighbour nodes, by the direction of traversal (clockwise or anticlockwise). Hence, we can replace each logical link in the ring with a left and a right link, in opposite directions, giving rise to the concept of left and right neighbours. The left neighbour of the node can be assigned the priority for taking over routing in case of node failure detection. As the overlay contains links other than left and right neighbours, depending on other requirements and formation steps, an irregular chordal ring[11], as illustrated in Figure 2, promises to be a good choice for the overlay structure.

Failure detection and alternative routing are taken care of in the algorithms to be executed by the overlay nodes, through a small and independent set of messages.

The second requirement (see R.2) is that the overlay node and its neighbours should have paths existing to the destination node, unless three or more nodes or links in the underlying network fail. This would be met if we have an overlay network with an availability degree of three.
2.3. Availability

The degree of availability of an overlay network [7] has been defined as the minimum number of underlay node disjoint paths that exist between any two overlay nodes in the network, taken over all pairs of overlay nodes in the network. It can be shown that if the chordal ring (described in Section 2.2) is constructed with the nodes of an overlay network with an availability degree of three, then every node and its neighbours will have three pairwise node disjoint paths to every other node in the overlay.

Let a, b and c be three neighbouring nodes in the chordal ring (shown in Figure 2), and let f be any other overlay node. Then consider the failure of two underlay nodes (other than that corresponding to f). There will be at least one path remaining from \{a, b, c\} to f, as each of them has three node disjoint paths to f in the underlay.

Overlay networks with an availability of degree three can be self configured using Trimarg [5].

2.3.1 Trimarg

Trimarg [5] is an algorithm for formation of overlays with an availability degree of three. Trimarg overlays are formed from an initial stellar overlay network of four broker nodes. It is based on the principle that if a new overlay node joins an existing overlay having degree of availability of three with node disjoint paths to three different expander nodes (underlay nodes having degree greater than two) [6], then the resulting overlay also satisfies availability of degree three. The construction technique is self organizing and the detailed proof is given in [5]. Hereafter, in this paper, when we refer to overlays with a degree of availability of three, we mean Trimarg generated overlays.

3. Underlay Aware Chordal Rings

The chordal ring used in our technique is built over an underlay aware overlay with an availability degree of three. We chose a chordal ring which has two neighbours for alternate storage of routing table of a node, designated as left and right neighbours. The chordal ring formed by our algorithm is illustrated in Figure 1. The left and right links are directional, but the chordal edges are bidirectional.

The chordal ring[10] is an irregular chordal ring which has an embedded simple ring in it. Every node in the chordal ring has an id, which determines its position in the ring. The ids range from 0 to \(2^{\text{max}} - 1\), where \(2^{\text{max}}\) is the maximum number of nodes possible. The node id’s increase in the left direction (except for the change from the last node to node zero, along left pointers). The node id’s similarly decrease along right links, except for the right link from zero to the highest id node. The nodes in the initial stellar broker network are illustrated in Figure 3. They are given id’s and linked to form a chordal ring of four nodes as shown in the Figure 4. Every node in the overlay with degree of availability three has three overlay links (underlay node disjoint) to three different overlay nodes in the network. In our algorithm, an overlay node executes a Join_Chordal_Ring routine to join the chordal ring and get a chordal ring node id. In this routine, it contacts the nearest of the three overlay neighbours, to request the neighbour to join the node in the chordal ring. The neighbour checks the path information and decides whether it should be joined as left neighbour or right neighbour. If the new node is to be joined as the left neighbour, then the node assigns its id, sends the routing table, and informs the original left neighbour to make the new node as its right neighbour. But if the new node is to be joined as right neighbour by the broker, it simply forwards the request to the right neighbour and the right neighbour makes the new node join as its left neighbour. This convention is fixed so as to disallow duplication of ids, and wrong formation of the ring in the distributed network.

In the next section, we give a detailed description of the self organizing asynchronous distributed algorithms for the formation and maintenance of the chordal ring in the event of dynamic and concurrent or sequential joining and leaving.
A brief description of the messages used in this algorithm is given below:

1. **New_Neighbour(B,q)**
   This is the message sent from the new node B to the neighbour node q, which node B has found to be the nearest to it.

2. **Assign_id(q, A, id, Chordal_Neighbours, Routing_Tables)**
   Message sent to the new broker node from node A (can some times be node q itself) with the new id assigned to the node, address of the two chordal ring neighbours and the routing tables of those two neighbours.

3. **Leave_Chordal_Ring_Request(Left_Neighbour, Right_Neighbours, Routing_Table, Right_Neighbour_id)**
   Message sent from the leaving broker to its left neighbour. Right neighbour’s id and routing table copy are also sent along in this message.

4. **Forward_Request(Neighbour)**
   Forwards the request message to the Neighbour.

5. **Forward_Join_Request (Neighbour _node, Source, Currentnode, Requestid)**
   Currentnode forwards the Join_Request from Source to the Neighbour_node with a Requestid assigned.

6. **Confirm_Leave_Chordal_Ring (Neighbour)**
   Neighbour sends the confirmation message allowing the sent broker to leave the chordal ring.

7. **New_id(Current_node_id, Left_Neighbour, RT)**
   Current_node sends the new id, Left_Neighbour’s id and the Routing Tables to the newly joining node.

8. **Routing_Tables_Ack**
   Acknowledgement message sent on receiving the routing table.

9. **New_right(Left_Neighbour,newid, Newrightneighbour)**
   Informs the current Left_neighbour of the node that there is a new right neighbour Newrightneighbour for that node in the chordal ring with id newid.

10. **Forward_Left_Request(L)**
    Forwards the request to the left neighbour.

11. **Request_Ackmnt()**
    Acknowledgement message sent on receiving forwarded requests for joining of a new broker.

12. **Processed_Request(Request_id, X, B, Broker_id, Routing_table)**
    This indicates that Requestid has been processed and node B is assigned the Broker_id and indicates that X should add B as its right neighbour. Routing_table is the current routing table of the new node B.

13. **Forwarded_Proc_Request(Requestid)**
    This message is sent by a node X which does not have the joining request Requestid in its right Queue because X joined at a later point of time than the request was enqueued. The node which receives this deletes the Requestid from its right Queue if it is present else forwards it to the left neighbour.

14. **Confirm_Leave_Chordal_Ring()**
    This is sent by a right neighbour to a leaving left neighbour after taking over its routing.

15. **Forward_Leaving_Message()**
    This is sent by a leaving node to its left neighbour on receiving a Leave_Chordal_Ring request from its right neighbour.

16. **Routing_Table_Updates()**
    This is sent periodically by each node to its left and right neighbours when its routing table is changed in order to update the neighbours’ stored routing tables.

17. **Refresh()**
    This is sent periodically by each node to confirm its presence to the neighbours.

18. **Urgent_Confirm()**
    This is sent by a node to its left neighbour if it does not get the Refresh message from the left neighbour. This is for the purpose of initiating a timeout interval to confirm the left neighbour’s presence before taking over its routing activities.

19. **Urgent_up()**
    This is sent as a reply to Urgent_Confirm message. If this message is received by the right neighbour, it will not start routing on behalf of the sending node.
4.1. A New Broker Joining the chordal ring

Algorithm 1 Join_Chordal_Ring(B)
1. Find the nearest neighbour (say q) in the set of 3 neighbours established.
2. Send New_Neighbour(B, q)
3. While not (Receive(Assign_Id (q, A, id, chordal_neighbour, routing_tables))
   Wait;
/* receives an assigned id message from node A, which may or may not be from node q */
4. save (routing_tables)
5. update (neighbourlist)
6. start (execute_broker)

4.2. An Old Broker Leaving

Algorithm 2 Leave_Chordal_Ring()
1. Send Leave_Chordal_Ring_Request (Left_Neighbour, Right_Neighbours_RoutingTable, Right_Neighbourid)
2. Send Leave_Chordal_Ring_Request (Right_Neighbour, Left_Neighbours_RoutingTable, Left_Neighbourid)
3. Leaving = True
4. if receive Confirm_Leave_Chordal_Ring(left) and Confirm_Leave_Chordal_Ring(right) and empty(queue) then stop (Execute_Broker(X))
/* An executing broker has to in addition to other tasks, process joining requests and add new brokers to the chordal ring */

4.3. Maintenance

Algorithm 3 Execute_Broker()
1. while true {
2. if receive(New_Neighbour(B, x)) then
3. if leaving=true then
4. Forward_Request(Left_Neighbour)
5. else compare (PathTo_B, PathTo_Left_Neighbour)
6. compare (PathTo_B, PathTo_Right_Neighbour)
7. if closer (B, Left_Neighbour)
8. then chosendir=left
9. else chosendir=right
10. R ← Right_Neighbour
11. L ← Left_Neighbour
12. Assign_Requestid
13. if chosendir=right
14. then enqueue (RightQ, Requestid, B, R)
15. send Forward_Join_Request(R, X, B, Requestid)
/* If the new node is to be inserted into the right side of the current node then the right neighbour is instructed to make the new node join as its left neighbour */
16. if chosendir=left
17. then enqueue (LeftQ, Requestid, B, L)
18. if not empty (LeftQ)
19. then S ← first (LeftQ)
/* If the left neighbour of the node which is trying to insert the new node has not changed since the time the request was enqueued, or if changed, the same node is the better (closer) node, it will insert the new node as its left neighbour else the request is forwarded to the better (closer) node */
20. if (Left_Neighbour = L(S)) OR Better_Neighbour (Left_Neighbour, B)
21. then calculate newid ← (id + id(left))/2
22. send New_id (B, newid, Left_Neighbour, Routing_Table)
23. while not Receive (Routing_Tables_ackmtn) wait
/* Current node informs its left neighbour that it has a new right neighbour to be joined */
24. send New_Right (Left_Neighbour, newid, Neighbourid)
25. deletequeue (LeftQ)
26. else Forward_Left_Request (Left_Neighbour)
27. while not Receive (Request_Ackmtn) wait
28. deletequeue (LeftQ)
/* When a node receives a message from its right neighbour, saying that it has a new right neighbour (the newly joined node B), then the node sets its right neighbour as B, stores the routing table, and deletes the corresponding joining request from its right queue if present. If the request is not present in its right queue, it must be because it joined after the joining request for B was enqueued, hence its left neighbour, or a node further left would be having the request in its right queue. So, the node forwards the message to its left neighbour. It gets further forwarded if needed till it reaches the node which has the corresponding request in its right queue */
29. if Receive (Processed_Request (Requestid, X, B, Broker_id, Routing_Table)) then
30. Right_Neighbour ← B
31. Store Routing_Table
32. if present (Requestid, RightQ)
33. then Deletequeue (RightQ)
34. else send Forwarded_Proc_Request (Requestid, Left_Neighbour)
35. if Receive (Forwarded_Proc_Request (Requestid))
36. then present (Requestid, RightQ)
37. then delete (RightQ (Requestid))
38. else send Forwarded_Proc_Request (Requestid, Left_Neighbour)
39. if Receive (Leave_Chordal_Ring) AND not (Leaving=true)
40. then Store Routing_Tables
41. if Receive (Leave_Chordal_Ring) AND Leaving=true
42. then Forward leaving message to Left_Neighbour
43. if (Receive (leaving(x)) AND (x = Left_Neighbour))
44. then Send Confirm_Leave_Chordal_Ring
45. if Receive New_Right (Left_Neighbour,
Routing

62. then start timeout
61. if (not(Routing
60. while (not
/*from stored routing table*/
57. start routing(Routing_Table)
56. if timeout
55. then Send
/* Send refresh signals to neighbours to confirm its presence*/
54. then Send(Routing_Table Updates) then Update Routing Tables
53. if not (Receive
52. then Send(Routing_Table)
51. if (Receive(Urgent_Confirm(x))
50. then Send(Routing_Table)
49. Send(Routing_Table)
48. if (Receive(Routing)
47. then Forward(Routing_Table)
46. then Send(Routing_Table)
45. then Forward(Routing_Table)
44. if (Receive(Routing)
43. then Send(Routing_Table)
42. then Send(Routing_Table)
41. then Send(Routing_Table)
40. then Send(Routing_Table)
39. then Send(Routing_Table)
38. then Send(Routing_Table)
37. then Send(Routing_Table)
36. then Send(Routing_Table)
35. then Send(Routing_Table)
34. then Send(Routing_Table)
33. then Send(Routing_Table)
32. then Send(Routing_Table)
31. then Send(Routing_Table)
30. then Send(Routing_Table)
29. then Send(Routing_Table)
28. then Send(Routing_Table)
27. then Send(Routing_Table)
26. then Send(Routing_Table)
25. then Send(Routing_Table)
24. then Send(Routing_Table)
23. then Send(Routing_Table)
22. then Send(Routing_Table)
21. then Send(Routing_Table)
20. then Send(Routing_Table)
19. then Send(Routing_Table)
18. then Send(Routing_Table)
17. then Send(Routing_Table)
16. then Send(Routing_Table)
15. then Send(Routing_Table)
14. then Send(Routing_Table)
13. then Send(Routing_Table)
12. then Send(Routing_Table)
11. then Send(Routing_Table)
10. then Send(Routing_Table)
9. then Send(Routing_Table)
8. then Send(Routing_Table)
7. then Send(Routing_Table)
6. then Send(Routing_Table)
5. then Send(Routing_Table)
4. then Send(Routing_Table)
3. then Send(Routing_Table)
2. then Send(Routing_Table)
1. then Send(Routing_Table)
0. then Send(Routing_Table)"

4.4. Complexity Analysis

The algorithms have been analyzed with respect to time, space and message complexities.

4.4.1 Time Complexity

Let path_length denote the average path length counting expander nodes from broker to its neighbour.

Join_Chordal_Ring: The time complexity is the time required for the comparison of the three obtained paths, which is O(path_length).

Leave_Chordal_Ring(): Time complexity depends on the wait time, which is O(path_length).

Execute_Broker(): Path comparisons cause the time complexity of this routine to be O(path_length).

4.4.2 Message Complexity

Join_Chordal_Ring and Leave_Chordal_Ring(): The message complexity is the number of messages exchanged which is O(1).

Execute_Broker(): If no forwarding is required Execute_Broker() sends a fixed number of messages. If forwarding is required, it just sends a fixed number of extra messages. Hence, message complexity is O(1).

4.4.3 Space Complexity

Join_Chordal_Ring() and Leave_Chordal_Ring() have a space complexity O(routing_table.size).

Execute_Broker() has a space complexity determined by the O(queue.size+ routing_table.Size). Queue.size is determined by the rate of broker-joining per broker node.

5. Proof of Correctness

For the chordal ring overlay to be correct, it should satisfy three properties always, which we state in the form of the following lemma.

Lemma 1: In the overlay formed by the executions of the distributed algorithms
(i) At any point of time there is only one node with a particular id executing Execute_Broker.
(ii) For any node b executing Execute_Broker, there exists a node whose left neighbour is b, and a node whose right neighbour is b.
(iii) If a node b which was executing Execute_Broker fails then within a fixed time interval τ the routing activity for b is taken up by the network, unless the left and right neighbour also fail, or at least two more nodes in the underlay fail.

Inductive proof for the lemma:

Base: Initially the three degree of availability overlay of stellar broker nodes is configured to be a chordal ring. By construction, parts (i) and (ii) of Lemma I are satisfied. Stellar broker nodes[5] are assumed to be nodes that never fail. Hence the property (iii) is also satisfied trivially.

In the inductive step, we show that augmentations or deletions to the network under concurrent or sequential executions of the routines in a distributed manner by different
nodes also results in an overlay with the above properties. Thus it demonstrates the self organization of the overlay.

**Inductive step:** For an overlay network $N$ with a chordal ring, satisfying the Lemma I, a new network $N'$ obtained by the

(a) Sequential executions of $\text{Join}_\text{Chordal Ring}$ or $\text{Leave}_\text{Chordal Ring}$ by different broker nodes

(b) Concurrent executions of $\text{Join}_\text{Chordal Ring}$ or $\text{Leave}_\text{Chordal Ring}$ by different brokers

(c) Concurrent executions of $\text{Join}_\text{Chordal Ring}$ and $\text{Leave}_\text{Chordal Ring}$ by different brokers also satisfies properties (i), (ii) and (iii) stated in Lemma I.

**Proof:**

**Property (i):**

**Case a:** If nodes join to two different nodes then id’s are different by the method of calculation of ids in line 21 of the $\text{Execute}_\text{Broker}$ algorithm. As the new id is the mean value of a node’s id and its left neighbour’s id. If they join the same broker, they will be given ids according to the queue order, the ids will be different, as evident from line 21. Leaving brokers do not change existing ids.

**Case b:** The queue ensures that even if $\text{Join}_\text{Chordal Ring}$ executed by the brokers are concurrent they will get different ids because at a particular time, a particular id can be assigned by only one node, and that node will assign ids only in sequence due to the queue. Leave chordal ring executions do not change the ids of existing nodes.

**Case c:** If the $\text{Join}_\text{Chordal Ring}$ and $\text{Leave}_\text{Chordal Ring}$ requests are sent to different brokers, then as a unique id is assigned only by one broker, the condition remains satisfied, and broker leaving does not change any ids in the remaining network. If $\text{Leave}_\text{Chordal Ring}$ and $\text{Join}_\text{Chordal Ring}$ are received and processed by the same broker ($\text{Left}_\text{neighbour}$ leaving and new $\text{Left}_\text{neighbour}$ joining), either $\text{Leave}_\text{Chordal Ring}$ is processed first, or $\text{Join}_\text{Chordal Ring}$. If $\text{Leave}_\text{Chordal Ring}$ is processed first, it immediately sends the acknowledgement and activates routing table, thereby the $\text{left neighbour}$’s id is free. When $\text{Join}_\text{Chordal Ring}$ is processed, the new id is assigned according to the currently computed id, which may (or may not) be the same as the id of the broker that left. In either case, property (i) is maintained.

**Property (ii):**

**Case a:** For all new nodes joining, if they join through different existing brokers, they communicate and establish left and right neighbour nodes before doing $\text{Execute}_\text{Broker}$. If they join the same node, again the processing of one is complete before the processing of the next starts, hence the condition is always maintained. A node that leaves, stops routing only after receiving acknowledgement from its neighbours, which happens only after they get information about their new neighbours.

**Case b:** Concurrent executions, even if they are dealt with by the same broker, are processed sequentially due to the queue, and hence it is ensured that both get their left and right neighbours before the next one is processed.

**Case c:** By the reasoning used for property (i), the nodes would have a unique left and right neighbour also, as the processing is sequential.

**Property (iii):**

New broker nodes joining the network start $\text{Execute}_\text{Broker}$ only after their neighbour’s routing table is made available to them. Periodic refresh updates routing tables. As the resulting overlay has an availability of degree three [5], which means that between every pair of overlay nodes, there exist at least three pairwise node disjoint paths. Hence, if a node fails and its $\text{Left}_\text{neighbour}$ (or $\text{Right}_\text{neighbour}$ fails), unless $\text{Right}_\text{neighbour}$ (or $\text{Left}_\text{neighbour}$) also fails, routing will be handled by $\text{Right}_\text{neighbour}$ (or $\text{Left}_\text{neighbour}$) as these are uniquely defined and have the routing table, by the property (ii). If both neighbours fail then three broker nodes in total have failed, and only in that condition, routing will not be handled. If only two overlay nodes fail, the routing will occur unless the physical path from the third broker to the destination node contains the failed overlay nodes. As the overlay network guarantees three node disjoint paths, an overlay link will not be reached only if a physical node in the third path also fails. Thus, unless three or more nodes fail, routing is guaranteed to occur. Nodes leaving the overlay network, also guarantee exchange of routing tables before leaving.

6. Scalability Evaluation

We present some simulation results meant for demonstrating the scalability of our algorithm in terms overlay network sizes. Details about the simulation framework used, and simulations for testing properties other than scalability are not included in this paper due to space constraints. Moreover, the primary focus of the paper is on the self organizing nature of the self healing overlay, which has been theoretically proved. The $\text{Join}_\text{Chordal Ring}$, $\text{Leave}_\text{Chordal Ring}$ and $\text{Execute}_\text{Broker}$ routines are simulated, using a simulation framework extended from DSSIM [12]. The brokers to join are randomly selected from the 10,000 physical nodes in the physical network generated using BRITE[8] internet topology generator, and those to leave are randomly selected from existing broker nodes.
Time: The simulated time taken for the Join_Chordal_Ring and Leave_Chordal_Ring operations are measured and reported in Figure 5 and Figure 6. It is verified that the time to join and leave are independent of overlay sizes.

Messages: The number of messages taken for the Join_Chordal_Ring and Leave_Chordal_Ring operations are measured and reported in Figure 7 and Figure 8 respectively. The message counts for the operations are verified to be independent of the size of the broker network.

Space Requirement: The space requirement of broker nodes execution in the presence of new brokers joining and brokers leaving is measured and reported in Figure 9. The average space for broker joining decreases with increasing size of the broker network.

Discussion: Assuming routing tables of a node to be of a constant size, the space requirement depends on the queue sizes in the broker nodes. As our complexity analysis shows, the queue sizes would depend on the rate of broker arrival. As we assume a uniform arrival rate, the space requirement per broker node reduces as the size of the broker network increases. Hence space requirement per broker node.
reduces as broker network size increases, provided broker arrival rate is uniform.

7. Related Work

Baldoni et al.,[1] present an approach for efficient data dissemination through a self organizing broker overlay. However they assume that the tasks related to overlay topology maintenance such as insertion of new brokers or topology repairing after broker failures are carried out by a human operator. Michael A. Jaeger [4] handle the problem of fault handling in publish/subscribe systems with algorithms for self stabilizing content based routing and with a self stabilizing broker overlay network. [4] addresses the issue of the routing tables stored at the event brokers getting corrupted due to transient failures. The basic idea adopted for this is renewal of the routing table entries before the leasing period expires. This research work is specific to content based routing. Moreover the issue of underlay awareness is not considered. Structured peer to peer overlays such as Chord [15], Pastry [14] and CAN [13] are built using distributed hash tables. These structured overlays do not address the issue of underlay awareness. 

Siena[2], Hermes[12], and REBECA[3] are some of the well known event based middleware prototypes. We have studied these prototypes mainly focussing on the overlay topology used for the event broker network, support for self configuration, self healing and reliability in delivery of the event notifications. We find these approaches have not considered underlay node failure resilience as a major issue. However failures are common in networks and self management and dynamic adaptation to these failures remain as core design issues which are inadequately addressed. Gero et.al[9] discuss how publish/subscribe systems can be made self-stabilizing using content-based routing. The main idea adopted in [9] is that when the time elapsed between consecutive faults is long enough, corrupted parts of the routing tables are removed whereas the correct parts are refreshed in time and missing parts inserted. This work addresses faults in general and bundles the issue with content based routing. Zhenyu et.al[17] propose a self stabilizing publish/subscribe protocol which can recover from any arbitrary data corruption at any number of nodes. However this work is not addressing the problem of ensuring event notification delivery when the link/ node failures occur in the network. Hence, we conclude that current research in the area of peer to peer overlay networks has not considered the issue of ensuring reliable data delivery in the context of failures and self configuration as core design issues.

8. Conclusion

In this paper, we have presented an overlay network which is self healing for upto two overlay or underlay node failures, and self organizing. The algorithm and concepts are illustrated using Event Broker Networks as a backdrop. Overlay nodes can dynamically join or leave the network, but they ensure that the self healing properties of the network are maintained in the resultant network. To the best of our knowledge, this is the first self organizing overlay with self healing properties for underlay as well as overlay node failures. Our main contributions can be summarized as (i) An underlay aware chordal ring overlay design with reliability of data delivery guarantees (ii) Asynchronous distributed algorithms for the building and maintaining the self organizing network. Our provably correct algorithms are efficient in terms of time space and message complexities, and have been demonstrated to be scalable to large overlay sizes with simulation studies. The solution presented in this paper can be extended to higher reliability guarantees using chordal rings of higher degree on a higher availability overlay. Routing algorithms exploiting the reliability guarantees and multiple independent paths between overlay nodes can be developed for efficient data routing.

References


