A relaxed-ring for fault-tolerant structured overlay networks

Abstract

One of the main problems of peer-to-peer systems using ring topology is to guarantee lookup consistency in presence of multiple joins, leaves and failures of network nodes. Lookup consistency and fault-tolerance are considered crucial properties for building applications on top of peer-to-peer networks. We introduce a novel relaxed-ring architecture where leaves and failures are considered as the same event. This approach allows us to provide a fault-tolerant join algorithm and a self-organizing ring in all cases. The relaxed-ring maintenance algorithms do not prevent the network from answering lookup requests while nodes are joining or leaving. Limitations related to failure handling are formally identified, providing strong guarantees to develop applications on top of the relaxed-ring architecture. Besides permanent failures, the paper analyses temporary failures and broken links, which are often ignored.

1. Introduction

Building decentralised applications requires several guarantees from the underlay peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Structured overlay network providing a Distributed Hash Table (DHT) using Chord-like ring topology [10] are a popular choice when the application needs efficient routing, lookup consistency and accessibility of all resources.

Despite the self-organizing nature of the ring architecture, its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents temporary inconsistency with massive peers joining the network, even in fault-free systems. A stabilisation protocol must be run periodically to fix these inconsistencies. One possible solution is presented by DKS [5], offering an atomic join/leave algorithm based on a locking mechanism. Even when this approach offers strong guarantees, we consider locks extremely restrictive for a dynamic network based on asynchronous communication. Every lookup request must be suspended in presence of join or leave in order to guarantee consistency. Leaving peers are not allow to leave the network until they are granted with the relevant locks. Given that, peers crashing just leave the network without respecting the protocol of the locking mechanism breaking the guarantees of the system. Another critical problem for performance is presented when a peer crashes while some joining or leaving peer is holding its lock.

The problem lies in the fact that existing algorithms require the agreement of three nodes in order to perform a join or leave operation. Due to asynchronous communication these leads to inconsistencies in the ring. Introducing locks to solve the issue is not good for performance, and more relevant, it is not fault-tolerant. We have developed an algorithm that only needs the agreement of two nodes at each stage, which is easier to guarantee given point-to-point communication. This decision leads us to a relaxed-ring topology, simplifying the joining algorithm and becoming fault tolerant to permanent failures of nodes, and also to temporary failures and broken links, which are often ignored.

The next section gives more details of the related work. Section 3 describes the relaxed-ring architecture and its guarantees. We continue with further analysis of the topology ending with conclusions and future work.

2. The problem and related work

Chord is the canonical DHT with ring topology. Its algorithms for ring maintenance handling joins and leaves present well known problems of temporary inconsistent lookups, where more that one node appears to be the responsible for the same key. The network needs to trigger periodic stabilisation in order to fix the inconsistencies. Existing analyses conclude that the problem comes from the fact that joins and leaves are not atomic operations, and they always need the synchronization of three peers, which is hard to guarantee with asynchronous communication, which is inherent to distributed programming.

Existing solutions [7, 8] introduce locks in the algorithms in order to provide atomicity of the join and leave operations. Locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free systems, which is not realistic. A better solution is provided by DKS [5], simplifying the locking mechanism and

proving correctness of the algorithms in absent of failures. As we mentioned in the introduction, one of the main problem of DKS is that relies on peers gracefully leaving the ring, which is not efficient nor fault-tolerant.

3. P2PS's relaxed-ring

The relaxed-ring topology is part of the new version of P2PS [9], which is designed as a modular architecture based on tiers. The whole system is implemented using the Mozart-Oz programming system [4], where the lowest level tier implements a communication session between two nodes. This point-to-point communication is implemented using Oz ports, having similar semantics to Erlang [1]. On top of this layer we implement a multi-session tier to manage several connections for a node. Such multi-sessions will be used by every peer to contact its successor, predecessor and finger pointers. The next upper tier, which is the focus of this paper, is in charge of the maintenance of the relaxed-ring topology. This layer can correctly route lookup requests providing consistency. Other layers built on top of this one are in charge of providing efficient routing. On top of these layers, other tiers implements reliable message sending, broadcast/multicast primitives and naming services, providing sufficient support to build decentralised systems using services architectures such as P2PKit [6].

As any overlay network built using ring topology, in our system every peer has a successor, predecessor, and fingers to jump to other parts of the ring in order to provide efficient routing. Ring's key-distribution is formed by integers from 0 to N growing clockwise. For the description of the algorithms we will use event-driven notation. When a peer receives a message, the message is triggered as an event in the ring maintenance tier.

Range between keys, such as $(p, q]$ follows the key distribution clockwise, so it is possible that $p > q$, and then the range goes from p to q passing through 0. Parentheses '(' and ')' excludes a key from the range and, '[' and ']' includes it.

3.1. The relaxed-ring

As we previously mentioned, one of the problem we have observed in existing ring maintenance algorithms is the need for an agreement between three peers to perform a join/leave action. We provide an algorithm where every step only needs the agreement of two peers, which is guaranteed with a point-to-point communication. In the specific case of a join, instead of having one step involving 3 peers, we have two steps involving 2 peers. The lookup consistency is guaranteed between every step and therefore, the network can still answer lookup requests while simultaneous nodes

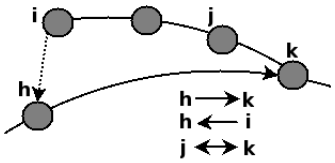

Figure 1. The relaxed-ring architecture

are joining the network. Another relevant difference is that we do not rely on graceful leaving of peers, because anywya, we have to deal with leaves due to network failures.

Our first invariant is that *every peer is in the same ring as its successor*. Therefore, it is enough for a peer to have connection with its successor to be considered inside the network. Secondly, the responsibility of a peer starts with the key of its predecessor plus 1, and it finishes with its own key. Therefore, a peer does not need to have connection with its predecessor, but it must know its key. These are two crucial properties that allow us to introduce the relaxation of the ring. When a peer cannot connect to its predecessor, it forms a branch from the *"perfect ring"*. Figure 1 shows a fraction of a relaxed ring where peer k is the root of a branch, and where the connection between peers h and i is broken.

Having the relaxed-ring architecture, we introduce a new principle that modifies the routing mechanism. The principle is that *a peer can never indicate another peer as responsible for a key*. This implies that even when the successor of a peer seems to be the responsible of a key, the request must be forwarded to the successor. Considering the example in figure 1, h may think that k is the responsible for keys in the interval $(h, k]$, but in fact there are three other nodes involved in this range. Note that the forwarding of a lookup request can be directed forward of backward with respect to the key distribution. We show later that this modification to the usual routing mechanism does not creates cycles and always converge.

3.2. Lookup

Before starting the description of the algorithms that maintain the relaxed-ring topology, we first define what do we mean by lookup consistency.

Def. *Lookup consistency implies that at any time there is only one responsible for a particular key k, or the responsible is temporary not available.*

Algorithm 1 describes the initial procedure of a node that wants to join the ring. First, it gets its own identifier from a random key-generator. In the implementation, identifiers also represent network references. For simplicity of the description of the algorithms, we will just use the key as identifier and as connection reference. Initially, the node does

not have a successor ($succ$), so it does not belong to any ring, and it does not know its predecessor ($pred$), so obviously, it does not have responsibilities. For resilient purposes, the node uses two sets: a successor list ($succlist$) and an old-predecessor sets ($predlist$). Having an access point, that can be any peer of the ring, the new peer triggers a lookup request for its own key in order to find its best successor candidate. This is quite usual procedure for several Chord-alike systems. When the responsible of the key contacts the new peer, the event $reply\_lookup$ is triggered in the new peer. This event will generate a joining message that will be discussed in section 3.3.

Algorithm 1 Starting a peer and the lookup algorithm

```
1:  procedure init(accesspoint) is
2:      self := getRandomKey()
3:      succ := nil
4:      pred := nil
5:      predlist := ∅
6:      succlist := ∅
7:      send ⟨ lookup | self, self ⟩ to accesspoint
8:  end procedure

9:  upon event ⟨ lookup | src, key ⟩ do
10:     if (key ∈(pred, self]) then
11:         send ⟨ reply_lookup | self ⟩ to src
12:     else
13:         p := getBetterResponsible(key)
14:         send ⟨ lookup | src, key ⟩ to p
15:     end if
16: end event

17: upon event ⟨ reply_lookup | i ⟩ do
18:     send ⟨ join | self ⟩ to i
19: end event
```

The $lookup$ event verifies if the current node is responsible for key. If it is not, it picks the best responsible for the key from its routing table, and forwards the request, passing the key and the original source src. Figure 2 depicts the algorithm for choosing the best responsible. The routing table of peer i consist on its successor, predecessor and two fingers, p and q. The requested key is k. To determine the closest peer to k, it is necessary to map the range of keys having k as the new 0, and $k-1$ as the new N. Then, the closest peer will be the one with the smallest mapped key. Note that since peer i, as every peer, has a successor and predecessor, there is always at least one peer closer to the target than the current one.

Operator **send** is a reliable point-to-point send. If the receiver presents a failure before the message arrives, the sender is notified. Then, to make the routing algorithm fault-tolerant, when **send** failes, the $lookup$ event must be

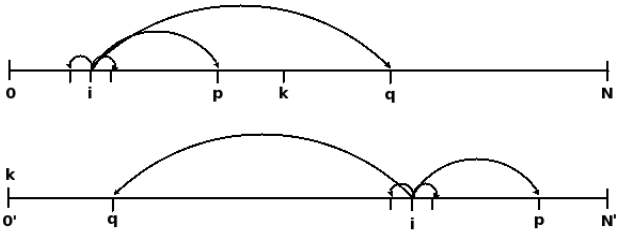

Figure 2. Choosing the better responsible.

triggered once again, finding a new best responsible if the peer did not became the responsible itself. If the failure was temporary, the message will be sent to the same peer.

Theorem 3.1 *The routing algorithm never introduces cycles, and always makes progress towards the key, eventually converging.*

Proof 1 *Every peer has a successor, a predecessor and a set of fingers. Successor and predecessor are enough to guarantee that after the mapping of key's domain, there will be always a peer closer to the target than the current peer. So the algorithm always makes progress. If several peers join or leave the ring while the routing is taking place, it does not affect the progress of the algorithm, because every peer will always have a closer peer, or it will become the responsible of the key, finishing the algorithm.*

3.3. The join algorithm

As we have previously mentioned, the relaxed-ring join algorithm is divided in two steps involving two peers each, instead of one step involving three peers as in existing solutions. The whole process is depicted in figure 3, where node q joins in between peers p and r. Following algorithm 1, r replies the lookup to q, and q send the $join$ message to r triggering the joining process.

The first step is described in algorithm 2, and following the example, it involves peer q and r. This step consists of two events, $join$ and $join\_ok$. Since this event may happen simultaneously with other joins or failures, r must verify that it has a successor, respecting the invariant that every peer is in the same ring as its successor. If it is not the case, q will be requested to retry later.

If it is possible to perform the join, peer r verifies that peer q is a better predecessor. For a regular join, function $betterPredecessor$ will just check if the key of the joining peer is in the range of responsibility of the current peer. If that is the case, p becomes the old predecessor and is added to the $predlist$ for resilient purposes. The $pred$ pointer is set to the joining peer, and the message $join\_ok$ is send to it.

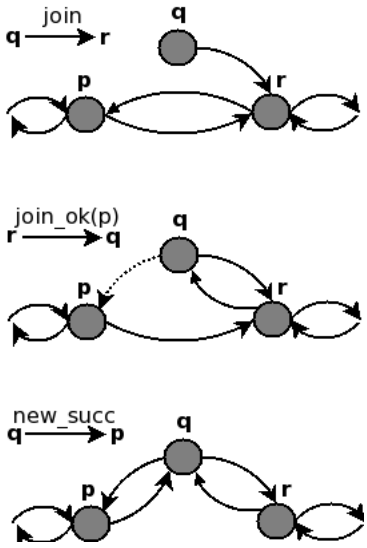

Figure 3. The join algorithm.

It is possible that the responsibility of r has change between the events $reply\_lookup$ and $join$. In that case, q will be redirected to the corresponding peer with the $goto$ message, eventually converging to the responsible of its key.

When the event $join\_ok$ is triggered in the joining peer q, the $succ$ pointer is set to r and $succlist$ is initialised. Then, q must set its $pred$ pointer to p acquiring its range of responsibility. At this point the joining peer has a valid successor and a range of responsibility, and then, it is considered to be part of the ring, even if p is not yet notified about the existence of q. This is different than all other ring networks we have studied.

Note that before updating the predecessor pointer, peer q must verify that its predecessor pointer is nil, or that it belongs to its range of responsibility. This second condition is only used in case of failure recovery and it will be described in section 3.5. In a regular join, $pred$ pointer at this stage is always nil.

Once q set $pred$ to p, it notifies p about its existence with message new_succ, triggering the second step of the algorithm.

The second step of the join algorithm basically involves peers p and q, closing the ring as in a regular ring topology. The step is described in algorithm 3. The idea is that when p is notified about the join of q, it updates its successor pointer to q (after verifying that is a correct join), and it updates its successor list with the new information. Functionally, this is enough for closing the ring. An extra event has been added for completeness. Peer p acknowledges its old successor r, about the join of q. When $joing\_ack$ is triggered at peer r, this one can remove p from the resilient $predlist$.

If there is a communication problem between p and q,

Algorithm 2 Join step 1 - adding a new node

```
1:  upon event ⟨ join | i ⟩ do
2:      if succ == nil then
3:          send ⟨ try_later | self ⟩ to i
4:      else
5:          if betterPredecessor(i) then
6:              oldp := pred
7:              pred := i
8:              predlist := {oldp} ∪ {predlist}
9:              send ⟨ join_ok | oldp, self, succlist ⟩ to i
10:         else if (i < pred) then
11:             send ⟨ goto | pred ⟩ to i
12:         else
13:             send ⟨ goto | succ ⟩ to i
14:         end if
15:     end if
16: end event

17: upon event ⟨ join_ok | p, s, sl ⟩ do
18:     succ := s
19:     succlist := {s} ∪ sl \ getLast(sl)
20:     if (pred == nil) ∨ (p ∈ (pred, self)) then
21:         pred := p
22:         send ⟨ new_succ | self, succ, succlist ⟩ to pred
23:     end if
24: end event

25: upon event ⟨ goto | j ⟩ do
26:     send ⟨ join | self ⟩ to j
27: end event
```

Algorithm 3 Join step 2 - Closing the ring

```
1:  upon event ⟨ new_succ | s, olds, sl ⟩ do
2:      if (succ == olds) then
3:          oldsucc := succ
4:          succ := s
5:          succlist := {s} ∪ sl \ getLast(sl)
6:          send ⟨ join_ack | self ⟩ to oldsucc
7:          send ⟨ upd_succlist | self, succlist ⟩ to pred
8:      end if
9:  end event

10: upon event ⟨ join_ack | op ⟩ do
11:     if (op ∈ predlist) then
12:         predlist := predlist \ {op}
13:     end if
14: end event
```

the event new_succ will never be triggered. In that case, the ring ends up having a branch, but it is still able to resolve queries concerning any key in the range $(p, r]$. This is because q has a valid successor and its responsibility is not shared with any other peer. It is important to remark the fact that branches are only introduced in case of communication problems. If q can talk to p and r, the algorithm provides a perfect ring.

No distinction is made concerning the special case of a ring consisting in only one node. In such a case, $succ$ and $pred$ will point to $self$ and the algorithm works identically. The algorithm works with simultaneous joins, generating temporary or permanent branches, but never introducing inconsistencies. Failures are discussed in section 3.5. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

Theorem 3.2 *The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.*

Proof 2 *Let us assume the contrary. There are two peers p and q responsible for key k. In order to have this situation, p and q must have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$. The join algorithm updates the predecessor pointer upon events $join$ and $join\_ok$. In the event $join$, the predecessor is set to a new joining peer j. This means that no other peer was having j as predecessor because it is a new peer. Therefore, this update does not introduce any inconsistency. Upon event $join\_ok$, the joining peer j initiates its responsibility having a member of the ring as predecessor, say i. The only other peer that had i as predecessor before is the successor of j, say p, which is the peer that triggered the $join\_ok$ event. This message is sent only after p has updated its predecessor pointer to j, and thus, modifying its responsibility from $(i, p]$ to $(j, p]$, which does not overlap with j's responsibility $(i, j]$. Therefore, it is impossible that two peers has the same predecessor.*

3.4. Resilient information

During the starting and join algorithms we have mentioned $predlist$ and $succlist$ for resilient purposes. The basic failure recovery mechanism is triggered by a peer when it detects the failure of its successor. When this happens, the peer will contact the members of the successor list successively. The objective of the $predlist$ is to recover from failures when there is no predecessor that triggers the recovery mechanism. This is expected to happen only when the tail of a branch has crashed. Section 3.5 gives more details about the recovery algorithms. Initially, we do not use extra fingers for recovery because it is not efficient. They may help to solve network partitioning, but we delegate this kind of recovery to upper layers of P2PS.

Algorithm 4 describes how the update of the successor list is propagated while the list contains new information. The predecessor list is updated only during the join algorithm and upon failure recoveries.

Algorithm 4 Update of successor list

```
1: upon event ⟨ upd_succlist | s, sl ⟩ do
2:     newsl := {s} ∪ sl \ getLast(sl)
3:     if (s == succ) ∧ (succlist ≠ newsl) then
4:         succlist := newsl
5:         send ⟨ upd_succlist | self, succlist ⟩ to pred
6:     end if
7: end event
```

3.5. Failure Recovery

In order to provide a robust system that can be used on the Internet, it is unrealistic to assume a fault-free environment or perfect failure detectors, meaning complete and accurate. We assume that every faulty peer will eventually be detected (strongly complete), and that a broken link of communication does not implies that the other peer has crashed (inaccurate). To terminate failure recovery algorithms we assume that eventually any inaccuracy will disappear (eventually strongly accurate). This kind of failure detectors are feasible to implement on the Internet.

Every node monitors the communication with every peer it is connected to. If a failure is detected, the $crash$ event is triggered as it is described in algorithm 5. The detected node is removed from the resilient sets $succlist$ and $predlist$, and added to a $crashed$ set. If the detected peer is the successor, the recovery mechanism is triggered. The $succ$ pointer is set to nil to avoid other peers joining while recovering from the failure, and the successor candidate is taken from the successors list. The function $getFirst$ returns the peer with the first key found clockwise, and removes it from the set. It returns nil if the set is empty. Function $getLast$ is analogue. Note that as every crashed peer is immediately removed from the resilient sets, these two functions always return a peer that appears to be alive at this stage. The successor candidate is contacted using the $join$ message, triggering the same algorithm as for joining. If the successor candidate also fails, a new candidate will be chosen. This is verified in the if condition.

When the detected peer p is the predecessor, no recovery mechanism is triggered because p's predecessor will contact the current peer. The algorithm decides a predecessor candidate from the $predlist$ to recover from the case when the tail of a branch is the crashed peer. We will not explore this case further in this paper because it does not violate our

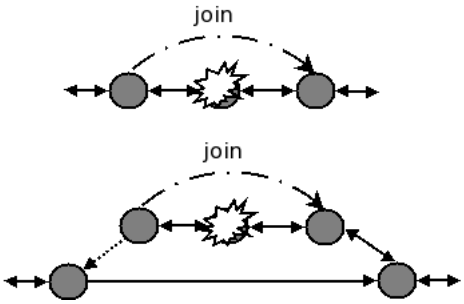

Figure 4. Simple crashes.

definition of consistent lookup. To solve it, it is necessary to set up a time-out to replace the faulty predecessor by the predecessor candidate.

When a link recovers from a temporary failure, the $alive$ event is triggered. This can be implemented by using watchers or a fault stream per distributed entity [3]. In this case, it is enough to remove the peer from the $crashed$ set. This will terminate any pending recovery algorithm. The faulty peer will trigger by itself the corresponding recovery events with the relevant peers.

Algorithm 5 Failure recovery

```
1:  upon event ⟨ crash | p ⟩ do
2:      succlist := succlist \ {p}
3:      predlist := predlist \ {p}
4:      chrased := {p} ∪ crashed
5:      if (p == succ) ∨ (p == succ_candidate) then
6:          succ := nil
7:          succ_candidate := getFirst(succlist)
8:          send ⟨ join | self ⟩ to succ_candidate
9:      else if (p == pred) then
10:         if (predlist ≠ ∅) then
11:             pred_candidate := getLast(predlist)
12:         end if
13:     end if
14: end event

15: upon event ⟨ alive | p ⟩ do
16:     crashed := crashed \ {p}
17: end event
```

Figure 4 shows the recovery mechanism triggered by a peer when it detects that its successor has a failure. The figure depicts two equivalent situations. The above one corresponds to a regular crash of a node in a perfect ring. The situation bellow shows that a crash in a branch is equivalent as long as there is a predecessor that detects the failure.

Having now the knowledge of the $crashed$ set, algorithm 6 gives complete definition of the function $betterPredecessor$ used in algorithm 2. Since the $join$ event is used both for a regular join and for failure recovery, the function will decides if a predecessor candidate is better than the current one if it belongs to its range of responsibility, or if the current $pred$ is detected as a faulty peer.

Algorithm 6 Verifying predecessor candidate

```
1: function betterPredecessor(i) is
2:     if (i ∈ (pred, self)) then
3:         return (true)
4:     else
5:         return (pred ∈ crashed)
6:     end if
7: end function
```

Knowing the recovery mechanism of the relaxed-ring, let us come back to our joining example and check what happens in cases of failures. If q crashes after the event $join$, peer r still has p in its $predlist$ for recovery. If q crashes after sending new_succ to p, p still has r in its $succlist$ for recovery. If p crashes before event new_succ, p's predecessor will contact r for recovery, and r will inform this peer about q. If r crashes before new_succ, peers p and q will contact simultaneously r's successor for recovery. If q arrives first, everything is in order with respect to the ranges. If p arrives first, there will be two responsible for the ranges $(p, q]$, but one of them, q, is not known by any other peer in the network, and it fact, it does not have a successor, and then, it does not belong to the ring. Then, no inconsistency is introduced in any case of failure.

Since failures are not detected by all peers at the same time, redirections during recovery of failures may end up in a faulty node. The correct version of the $goto$ event is described in algorithm 7. If a peer is redirected to a faulty node, it must insist with its successor candidate. Since failure detectors are strongly complete, the algorithm will eventually converge to the correct peer.

Algorithm 7 Modified goto

```
1: upon event ⟨ goto | p ⟩ do
2:     if (p ∉ crashed) then
3:         send ⟨ join | self ⟩ to p
4:     else
5:         send ⟨ join | self ⟩ to succ_candidate
6:     end if
7: end event
```

Figure 5 shows two simultaneous crashes together with a new peer joining before the peer used for recovery. If the recovery $join$ message arrives first, the ring will be fixed before the new peer joins, resulting in a regular join. If the new peer starts the first step of joining before the recovery, it will introduce a temporary branch because of its impossibility of contacting the faulty predecessor. When the recovery

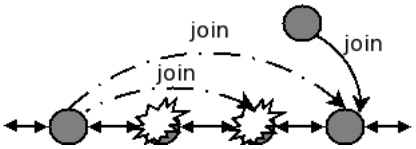

Figure 5. Multiple failure recovery and simultaneous join.

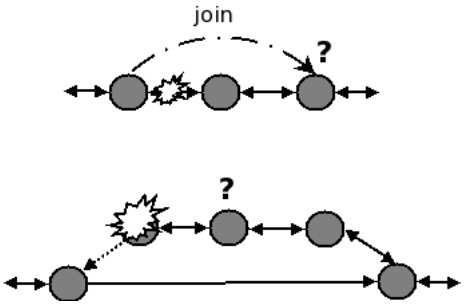

Figure 6. Broken link and failure of the tail of branch

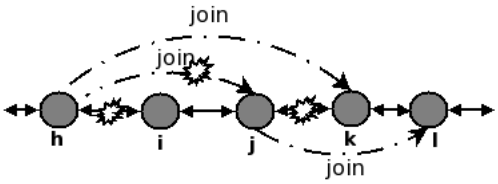

Figure 7. Network partition in a regular ring.

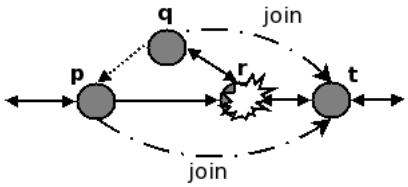

Figure 8. The failure of the root of a branch triggers two recovery events

$join$ message arrive, the recovering peer will contact the new joining peer, fixing the ring and removing the branch.

There are failures more difficult to handle than the ones we have already analysed. Figure 6 depicts a broken link and the crash of the tail of a branch. In the case of the broken link (inaccuracy), the failure recovery mechanism is triggered, but the successor of the suspected node will not accept the join message. The described algorithm will eventually recover from this situation when the failure detector reaches accuracy.

In the case of the crash of the node at the tail of a branch, there is no predecessor to trigger the recovery mechanism. In this case, the successor could use one of its nodes in the predecessor list to trigger recovery, but that could introduce inconsistencies if the suspected node has not really failed. If the tail of the branch has not really failed but it has a broken link with its successor, then, it becomes temporary isolated and unreachable to the rest of the network. Having unreachable nodes means that we are in presence of network partitioning. The following theorem describes the guarantees of the relaxed-ring in case of temporary failures with no network partitioning.

Theorem 3.3 *Simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition.*

Proof 3 *Every failure of a node is eventually detected by its successor, predecessor and other peers in the ring having a connection with the faulty node. The successor and other peers register the failure in the $crashed$ set, and remove the faulty peer from the resilient sets $predlist$ and $succlist$, but they do not trigger any recovery mechanism. Only the predecessor triggers failure recovery when the failure of its successor is detected, contacting only one peer from the successor list at the time. Then, there is only one possible candidate to replace each faulty peer, and then, it is impossible to have two responsible for the same range of keys.*

With respect to network partitions, there are two important cases we want to analyse and they are depicted in figures 7 and 8. Figure 7 depicts a situation that can happen in any system using ring topology, where nodes i and j become isolated from the rest of the ring. The situation goes as follows: peer h contacts peer j in order to recover from the inaccurate failure detected on peer i, but communication with j is also broken. Peer h contacts peer k, and k accepts the recovery because it has detected a failure in the communication with j. Due to the broken link between j and k, peer j contacts peer l, but this one does not accept the join message from j, because k appears to be alive. This introduce a temporary inconsistency with respect to range $(h, j]$. This temporary uncertainty has been proven by Ghodsi [5], and it is related to the proof provided in [2] about limitations of web services in presence of network partitioning.

Figure 8 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 3.3 is based on the fact that per every failure detected, there is only one peer that triggers the recovery mechanism. In the case of the failure of the root of a branch, peer r in the example, there are two recovery messages triggered by peers p and q. If message from peer q arrives first to peer t, the algorithm

handle the situation without problems. If message from peer p arrives first, the branch will be temporary isolated, behaving as a network partition introducing a temporary inconsistency. This limitation of the relaxed-ring is well defined in the following theorem.

Theorem 3.4 *Let r be the root of a branch, $succ$ its successor, $pred$ its predecessor, and $predlist$ the set of peers having r as successor. Let p be any peer in the set, so that $p \in predlist$. Then, the crash of peer r may introduce temporary inconsistent lookup if p contacts $succ$ for recovery before $pred$. The inconsistency will involve the range $(p, pred]$, and it will be corrected as soon as $pred$ contacts $succ$ for recovery.*

Proof 4 *There are only two possible cases. First, $pred$ contacts $succ$ before p does it. In that case, $succ$ will consider $pred$ as its predecessor. When p contacts $succ$, it will redirect it to $pred$ without introducing inconsistency. The second possible case is that p contacts $succ$ first. At this stage, the range of responsibility of $succ$ is $(p, succ]$, and of $pred$ is $(p', pred]$, where $p' \in [p, pred]$. This implies that $succ$ and $pred$ are responsible for the range $(p', pred]$, where in the worse case $p' = p$. As soon as $pred$ contacts $succ$ it will become the predecessor because $pred > p$, and the inconsistency will disappear.*

Theorem 3.4 clearly states the limitation of branches in the systems, helping developers to identify the scenarios needing special failure recovery mechanisms. Since the problem is related to network partitioning, there seems to be no easy solution for it. An advantage of the relaxed-ring topology is that the issue is well defined and easy to detect, improving the guarantees provided by the system in order to build fault-tolerant applications on top of it.

4. Future Work

The basic layers of P2PS providing point-to-point communication and the relaxed-ring maintenance are very stable. Our future work will be focused on the upper layers in order to deal with network partitioning. Apart from failure recovery, we are interested in building a service oriented architecture that will require a robust naming service and reliable broadcast.

5. Conclusion

In this paper we have presented a novel relaxed-ring topology for fault-tolerant and self-organising peer-to-peer networks. The topology is derived from the simplification of the join algorithm requiring the synchronisation of only two peers at each stage. As a result, the algorithm introduces branches to the ring. These branches can only be observed in presence of connectivity problems between peers, and help the system to work in realistic scenarios. The topology adds some complexity to the routing algorithm, but it keeps it correct and addressable. We consider this issue a small drawback in comparison to the huge gain in fault tolerance and cost-efficiency in ring maintenance.

The guarantees and limitations of the system are clearly identified and formally stated providing helpful indications in order to build fault-tolerant applications on top of this structured overlay network.

6. Acknowledgement

The authors would like to thank Kevin Glynn and Sebastián González for comments on this work. This research is mainly funded by EVERGROW (contract number:001935) and SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.

[2] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM Press.

[3] R. Collet and P. V. Roy. Failure handling in a network-transparent distributed programming language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[4] M. Consortium. The mozart-oz programming system. *http://www.mozart-oz.org*, 2007.

[5] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[6] K. Glynn. P2PKit: A services based architecture for deploying robust peer-to-peer applications. *http://p2pkit.info.ucl.ac.be/index.html*, 2007.

[7] X. Li, J. Misra, and C. G. Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.

[8] X. Li, J. Misra, and C. G. Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2):126–148, 2006.

[9] V. Mesaros, B. Carton, and K. Glynn. P2PS: A peer-to-peer networking library for mozart/oz. *http://p2ps.info.ucl.ac.be/index.html*, 2007.

[10] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.