Parallel Processing Letters

THE RELAXED-RING: A FAULT-TOLERANT TOPOLOGY FOR STRUCTURED OVERLAY NETWORKS

BORIS MEJÍAS AND PETER VAN ROY
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

Received May 2008
Revised July 2008
Communicated by P. Fragopoulou

ABSTRACT

Fault-tolerance and lookup consistency are considered crucial properties for building applications on top of structured overlay networks. Many of these networks use the ring topology for the organization or their peers. The network must handle multiple joins, leaves and failures of peers while keeping the connection between every pair of successor-predecessor correct. This property makes the maintenance of the ring very costly and temporarily impossible to achieve, requiring periodic stabilization for fixing the ring. We introduce the relaxed-ring topology that does not rely on a perfect successor-predecessor relationship and it does not need any periodic maintenance. Leaves and failures are considered as the same type of event providing a fault-tolerant and self-organizing maintenance of the ring. Relaxed-ring's limitations with respect to failure handling are formally identified, providing strong guarantees to develop applications on top of the architecture. Besides permanent failures, the paper analyses temporary failures and false suspicions caused by broken links, which are often ignored.

Keywords: peer-to-peer, network topology, relaxed-ring, self-configuration, fault-tolerance

1. Introduction

Building decentralized applications requires several guarantees from the underlaying peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Structured overlay networks using a Chord-like ring topology [15] are a popular choice when the application needs efficient routing, lookup consistency and accessibility of resources. According to [7], the ring topology is one of the most resilient to failures, and it is competitive with any other structured overlay network with respect to reaching any other node in a small amount of steps.

The ring topology offers many good properties as we just mentioned, but its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents *temporary inconsistencies* with peers massively join-

ing the network, even in fault-free environments, as we will discuss in section 4. A stabilization protocol must be run periodically to fix these inconsistencies increasing the load of the network. One possible solution is presented by DKS [6], 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 involving the critical range of keys must be suspended in presence of a join/leave event in order to guarantee consistency. Leaving peers are not allowed to leave the network until they are granted the relevant locks. Given that, crashing peers 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 with the maintenance of the ring is that routing algorithms and correctly assigning responsibilities over keys, rely on the perfect relationship between predecessor and successor. But, every join, leave or failure, brakes temporary this relationship. In order to solve this problem, existing algorithms require the agreement of three nodes to perform a *join* or *leave* operation. Managing three nodes at the same time can create unexpected problems, mainly because transitivity of communication cannot be assumed. If node a can talk to b, and b can talk to c, it does not mean that a can talk to c. This problem is equivalent to false suspicions of failure. Consider that a, b and c are talking to each other. Suddenly, the connection between a and c is broken. Peer a informs b about this failure, but b sees that c is alive, meaning that a falsely suspected of c. This is why algorithm based on the synchronized agreement of three nodes are not fault-tolerant. A recent work [14] conclude that lookup inconsistencies are mainly caused by false suspicions. Churn does not introduce inconsistencies if periodic stabilization is triggered often enough, but this very costly as it will be shown in Section 4.

The contribution of this work is an algorithm that only needs the agreement of two nodes at each stage of the maintenance of the ring. By working with only two nodes in every step we have arrived to the Relaxed-Ring topology, which allows a ring to be partially open. This approach simplifies the joining algorithm dividing it into two steps involving two peers each. Lookup consistency is guaranteed after every step. The algorithm provides a failure recovery mechanism where only two nodes interact in every step. Graceful leaves are consider a special case of failure, and therefore, they are equivalent events. Because of this, there is no need for a graceful leaving protocol. This is useful for end-users, because they can just shut down their application letting the network to handle their departure. Fault-tolerance is achieved at the level of permanent failures, temporary failures and false suspicions, which results from broken links, which are often ignored.

The Relaxed-Ring adds robustness to the network in presence of churn and failures. It simplifies the arrival and departure of peers by dividing these event into smaller and correct steps. Due to the relaxed topology, the routing performance is shortly degraded. On the other hand, there is no need for periodic maintenance of

the ring, because the Relaxed-Ring remains correct after every step, reducing the cost of maintenance. Part of this contribution has been published in [11], where the Relaxed-Ring was presented from the point of view of its design as a self-managing system. This work is focused on the correctness of the algorithm through analytical results, and an empirical validation with simulations.

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. Evaluation of the relaxed-ring is presented in section 4, ending with conclusions and future work.

2. The problem and related work

Chord is the canonical structured overlay network using ring topology. Its algorithms for ring maintenance handling joins and leaves have been already studied [6] showing problems of temporary inconsistent lookups, where more that one node appears to be the responsible for the same key. Peers need to trigger periodic stabilization in order to fix 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 [8, 9] introduce locks in the algorithms in order to provide atomicity of the join and leave operations, removing the need for a periodic stabilization. Unfortunately, locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free environments, which is not realistic. Another problem with these approaches is that they are not starvation-free, and therefore, it is not possible to guarantee liveness. A better solution using locks is provided by Ghodsi [6], using DKS [2] for its results. This approach is better because it gives a simpler design for a locking mechanism and proves that no deadlock occurs. It also guarantees liveness by proving that the algorithm is starvation-free. Unfortunately, the proofs are given in fault-free environments.

The DKS algorithm for ring maintenance goes already in the right direction because it request the locks of only two peers instead of three (as in [8, 9]). It works as follows. Every peer holds a lock that can be exclusively taken by any peer. The lock grants access to update the pointers of the peer. A joining/leaving peer needs to get its own lock and its successor's lock. Let us consider peer q joining in between p and r. Peer q first has to get its own lock and then the lock of r, its successor candidate. This is sufficient for q to update its predecessor and successor, to update r's predecessor, and to update p's successor pointer. Note that p cannot change its pointers because that would require getting r's lock, which is already taken by q. The situation of q leaving is analogous, with the difference that p show acquires q's lock in order to perform any action. This mechanism guarantees that if the relevant locks are acquired, the join/leave can be performed *atomically.*

One of the problem with the algorithm is that even when it only requires the

lock of two peers, it still requires the *atomic* update of the pointers of *three* peers. While this three changes are made, no lookup involving peers p, q or r is allowed. A more complex problem is that the algorithm relies on peers gracefully leaving the ring, which is neither efficient nor fault-tolerant. The algorithm becomes very slow if a peer holding a relevant lock crashes. How can the other peers continue? The same problem occurs if a locked peer stop responding. Another problems is that a joining peer q that acquires its own lock and r's lock, is not guaranteed to establish communication with p in order to change its successor pointer.

We are not aware of other approaches solving the problem of atomic join/leave with failure recovery, or other approaches targeting the elimination of periodic stabilization.

3. The Relaxed-Ring

The relaxed-ring topology has evolved from the Peer-to-Peer System (P2PS) [5], and it is implemented using the Mozart-Oz programming system [13]. 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 $N-1$ 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.

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. Lookup consistency is guaranteed after every step, therefore, the network can still answer lookup requests while simultaneous nodes are joining the network. Another relevant difference is that we do not rely on graceful leaving of peers. We treat leaves and failures as the same event. This is because failure handling already includes graceful leaves as a particular case.

Normally the overlay is a ring with predecessor and successor knowing each other. If a new node joins in between these two peers, it introduces two changes. The first one is to contact the successor. This step already allows the new peer to be part of the network through its successor. The second step, contacting the predecessor, will close the ring again. Following this reasoning, 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, excluding

predecessor's key, 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 t is the root of a branch, and where the connection between peers q and p is broken.

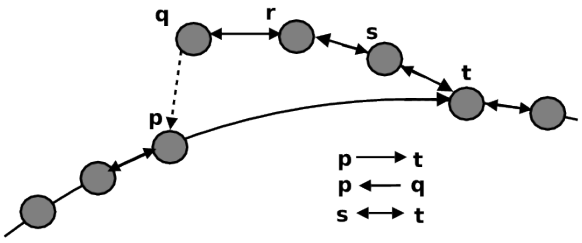

Fig. 1. A branch on the relaxed-ring created because peer q cannot establish communication with p. Peers p and s consider t as successor, but t only considers s as predecessor.

Having the relaxed-ring architecture, we introduce a new principle that modifies the routing mechanism of Chord. The principle is that *a peer p always forwards the lookup request to the possible responsible, even if p is the predecessor of such responsible.* Considering the example in figure 1, p may think that t is the responsible for keys in the interval $(p, t]$, but in fact, there are three other nodes involved in this range. In Chord, p would just reply t as the result of a lookup for key q. In the Relaxed-Ring, the p forwards the message to t. When the message arrives to node t, it is sent backwards to the branch, until it reaches the real responsible. Forwarding the request to the responsible is a conclusion we have already presented in [11], and it has been recently confirmed by Shafaat [14].

Introducing branches into the lookup mechanism modifies the guarantees about proximity offered by Chord. Reaching the root of a branch takes $O(log_k(n))$ hops as in Chord, because the root of the branch belongs to the *core-ring*. Then, the lookup will be delegated a maximum of b hops, where b corresponds to the size of the branch. Then, lookup on the relaxed-ring topology corresponds to $log_k(n) + b$. We will see in section 4 that the average value b is smaller than 1 for large networks.

Before continuing with the description of the algorithms that maintain the relaxed-ring topology, let us 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 many 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 3.1.

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*. Choosing the best responsible of a key follows the same mechanism as Chord, with the extra consideration of rooting to the branch when needed, as explained above. One way to decide that a lookup must jump into the branch is by adding a flag to the message called *last*. In the case of Figure 1, when p forwards the messages to t, it sets the flag to *true*. Then, the function *getBetterResponsible* will decide to forward to the predecessor, jumping in to the branch.

3.1. *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 2, 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.

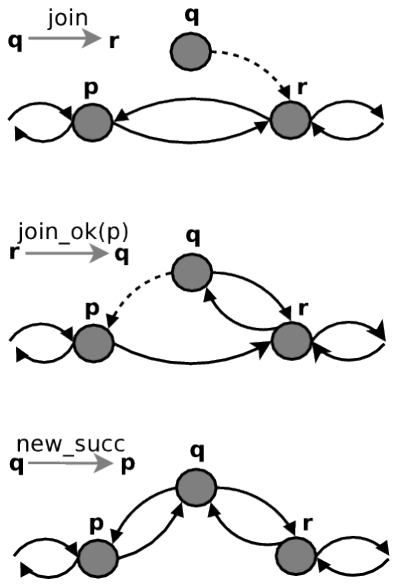

Fig. 2. The join algorithm.

If it is possible to perform the join, peer r verifies that peer q is better predecessor than p. Function *betterPredecessor* checks if the key of the joining peer is in the range of responsibility of the successor candidate. In the example, r verifies that $q \in (p, r]$. 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.

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 initialized. 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 3.3. 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 *join_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, 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 3.3. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

Theorem 1. The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.

Proof.

Let us assume the contrary. There are two peers p and q responsible for key k. If p and q have the same successor is not relevant, because both peers would forward the lookup to the successor, and the successor can resolve the conflict. The problem is when p and q have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$ introducing a inconsistency because of the overlapping or ranges. Let us see now that the algorithm prevents two nodes from

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
```

having the same predecessor. 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. □

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
```

3.2. *Reducing size of branches*

Let us consider again figure 1. If nodes keeps on joining as predecessors of peer t, the branch will increase its size, even if they could have a good connection with peer p. An improvement on the join algorithm will be that node t sends a *hint* message to node p avoid new joining peer. If p cannot talk to q, it does not mean that it can not talk to r or s. If the p can contact the *hinted* node, it will add it as its successor, making the branch shorter. This hint message will not modify the predecessor pointers of r or s. Peer t uses its *predlist* list for sending hints.

3.3. *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.

When the point-to-point communication layer detects a failure of one of the nodes, the *crash* event is triggered as it is described in algorithm 4. 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 variable

succ_candidate should be initialized to *nil* in the *init* event of Algorithm 1, but it was not included to avoid confusion at that part of the analysis of the algorithm. The real value is initialized at line 7 of the *crashed* event. 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 analogous. 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.

If a peer p detects that its predecessor *pred* has crashed, it will not trigger the recovery mechanism. It is *pred*'s predecessor who will contact p. In case that no peer contacts p for recovery, p could guess a predecessor candidate from its *predlist*, at the risk of breaking lookup consistency, but closing the ring again. We will not explore this case further in this paper because it does not violate our 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, but it would always take the risk of a reacting to a false suspicion.

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 [4]. 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 4 Failure recovery

```
1:  upon event ⟨ crash | p ⟩ do
2:      succlist := succlist \ {p}
3:      predlist := predlist \ {p}
4:      crashed := {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:      end if
10: end event

11: upon event ⟨ alive | p ⟩ do
12:     crashed := crashed \ {p}
13: end event
```

Figure 3 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

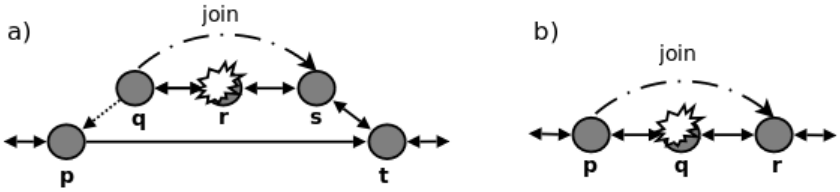

Fig. 3. Failures simple to handle: (a) In a branch, q and s detect that r has crashed. Only q triggers failure recovery. (b) Pers p and r detects q has crashed. Peer p triggers the recovery mechanism.

one corresponds to a regular crash of a node in a perfect ring. The situation below 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 5 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 5 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. In case of a network partition, these peers will get divided in two or three groups depending on the partition. In such case, they will continue with the recovery algorithm in their own rings. Global consistency is

impossible to achieve, but every ring will be consistent in itself.

Since failures are not detected by all peers at the same time, redirection during recovery of failures may end up in a faulty node. The correct version of the *goto* event is described in algorithm 6. 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 6 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 4 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 *join* message arrive, the recovering peer will contact the new joining peer, fixing the ring and removing the branch.

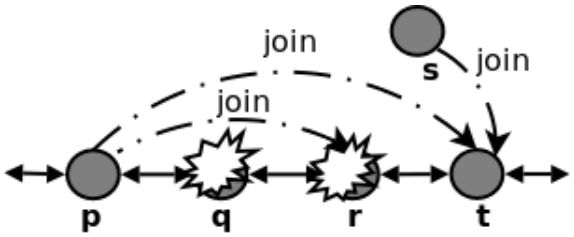

Fig. 4. Multiple failure recovery and simultaneous join. Peer p detects the crash of its successor q. First successor candidate r has also crashed. Peer p contacts t at the same time peer s tries to join the network. Both *join* messages are the same.

There are failures more difficult to handle than the ones we have already analysed. Figure 5 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 eventually provides accurate information.

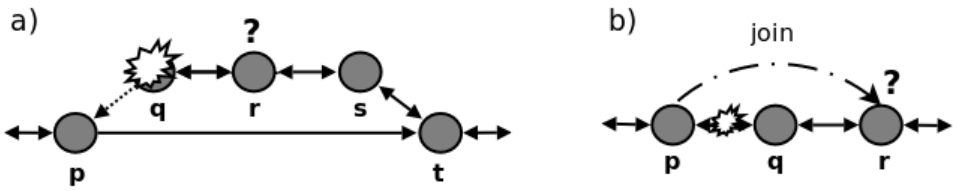

Fig. 5. Failures difficult to handle: (a) failure of the tail of branch, nobody is responsible for range $(p, q]$ (b) broken link generating a false suspicion of p about q.

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

Proof.

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. If a simultaneous join occurs (as in figure 4), there are two possible cases. If the recovery happens first, the join will just be as regular join. If the join happens first, the successor candidate will reject the recovery forwarding to the recovery to the new peer. This means that only one successor candidate for recovery will be contact at the time, preventing inconsistencies. □

The problem with respect to network partition is inherent to any overlay network, where a temporary uncertainty cannot be avoid, and some guarantees must be sacrificed. A deeper analysis is provided by Ghodsi [6], and it is related to the proof given in [3] about limitations of web services in presence of network partitioning.

Figure 6 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 2 is based on the fact that per every failure detected, there is

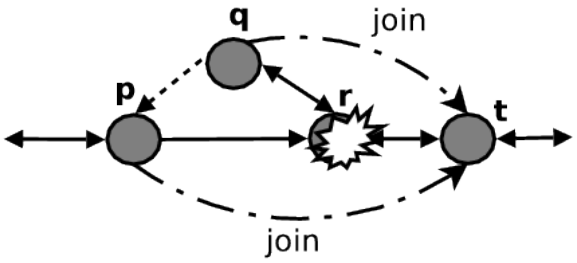

Fig. 6. The failure of the root of a branch triggers two recovery events

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.

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. 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 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.

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.

Algorithm 7 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 7 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
```

4. Evaluation

This section is dedicated to the evaluation of the relaxed-ring. We analyse four aspects: the *amount of branches* that can appear on a network, the *size of branches*, the *number of messages* generated by the ring-maintenance protocol, and the verification of *lookup consistency* on unstable scenarios. The evaluation is done using a simulator implemented in Mozart [13, 10], where every node run autonomously on its own lightweight thread. Nodes communicate with each other by message passing using ports. We consider that these properties make the simulator more realistic. Every network is run several times using different seeds for random number generation. Charts are built using the average values of these executions.

4.1. *Branches and messages*

Figure 7 shows the amount of branches that can appear on networks with 1000 to 10000 nodes. The coefficient c represents the connectivity level of the network, where for instance $c = 0.95$ means that when a node contacts another one, there is only a 95% of probability that they will establish connection. A value of $c = 1.0$ means 100% of connectivity. On that value, no branches are created, meaning that the relaxed-ring behaves as a perfect ring on fault-free scenarios. The worse case corresponds to $c = 0.9$. In that case, we can observe that the amount of branches

is less than 10% of the size of the network, as expected. Consider peers i and k, where i is the current predecessor of k. If they cannot talk to each either, k will form a branch. If another peer j joins in between i and k having good connection with both peers, the branch disappears.

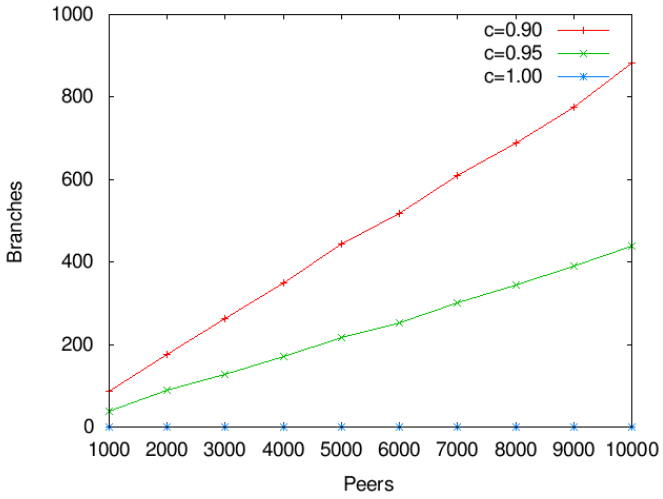

Fig. 7. Average amount of branches generated on networks with connectivity problems. Networks where tested with peers having a connectivity factor c, representing the probability of establishing a connection between peers, where $c \in \{0.9, 0.95, 1\}$.

On the contrary, if a node l joins the network between k and its successor, it will increase the size of the branch, decreasing the routing performance. For that reason, it is important to measure the average size of branches. If message *hint*, explained in section 3.2, works well for peer l, then, the branch will remain on size 1. Having this in mind, let us analyse figure 8. The average size of branches appears to be independent of the size of the network. The value is very similar for both cases where the quality of the connectivity is poor. In none of the cases the average is higher than 2 peers, which is a very reasonable value. If we want to analyse how the size of branches degrades routing performance of the whole network, we have to look at the average considering all nodes that belong to the core ring as providing branches of size 0. This value is represented by the curves *totalavg* on the figure. In both cases the value is smaller that 0.25. Experiments with 100% of connectivity are not shown because there are no branches, so the average size is always 0.

How many messages are exchanged by peers in order to maintain the relaxed-ring structure? How much is the contribution of the *hint* messages to the load in order to keep branches short? These questions are answered in figure 9. We can observe that the amount of messages increases linearly with the size of the network keeping reasonable rates. The fault-free scenario has no *hint* messages as expected, but the

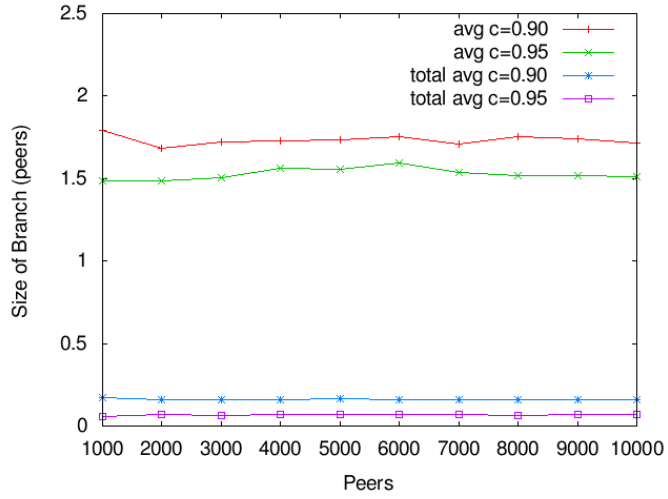

Fig. 8. Average size of branches depending on the quality of connections: *avg* corresponds to existing branches and *totalavg* represents how the whole network is affected.

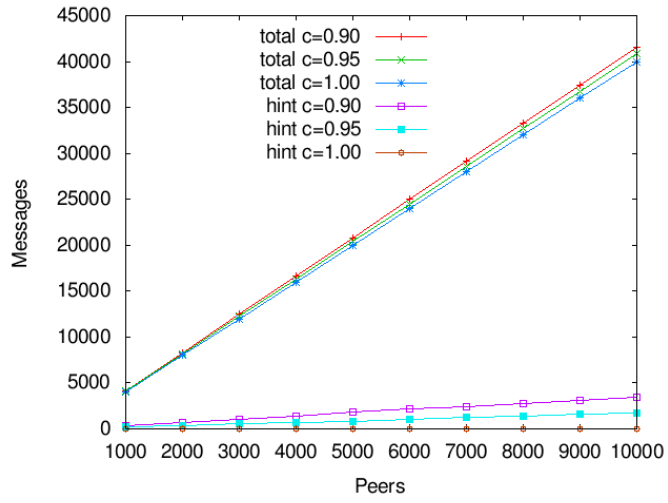

Fig. 9. Number of messages generated by the relaxed-ring maintenance. Three curves labeled *total* represent the total amount of messages exchanged between all peers depending on the connectivity coefficient. Curves labeled *hint* represent the contribution of *hint* messages to the total amount.

total amount of messages is still pretty similar to the cases where connectivity is poor. This is because there are less normal join messages in case of failures, but this amount is compensated by the contribution of *hint* messages. We observe anyway that the contribution of *hint* messages remains low.

4.2. *Comparison with Chord*

We have also implemented Chord in our simulator. Experiments were only run in fault-free scenarios with full connectivity between peers, and thus, in better conditions than our experiments with the relaxed-ring. Even though, we observed many lookup inconsistencies on high churn. To reduce inconsistency, we trigger periodic stabilization on all nodes at different rates. The best results appeared after triggering stabilization after the join of every 4 nodes. We call this value stabilization rate. As seen in figure 10, the largest the network, the less inconsistencies are found. An inconsistency is detected when two *reachable* nodes are signalized as responsible for the same key. We can observe that stabilization rates of 5 converges pretty fast to 0 inconsistencies. Stabilization every 6 new joining peers only converge on networks of 4000 nodes. On the contrary, rate values of 7 and 8 presents immediately a high and non-decreasing amount of inconsistencies. Those networks would only converge if churn is reduced to 0. These values are compared with the worse case of the relaxed-ring (connectivity factor 0.9) where no inconsistencies where found.

We have observed that lookup consistency can be maintained in Chord at very good levels if periodic stabilization is triggered often enough. The problem is that periodic stabilization demands a lot of resources. Figure 11 depicts the load related to every different stabilization rate. Logically, the worse case corresponds to most frequently triggered stabilization. If we only consider networks until 3000 nodes, it seems that the cost of periodic stabilization pays back for the level of lookup consistency that it offers, but this cost seems too expensive with larger networks.

In any case, the comparison with the relaxed-ring is considerable. While the relaxed-ring does not pass 5×10^4 messages for a network of 10000 nodes, a stabilization rate of 7 on a Chord network, starts already at 2×10^5 with the smallest network of 1000 nodes. Figure 11 clearly depicts the difference on the amount of messages sent. The point is that there are too many stabilization messages triggered without modifying the network. On the contrary, every join on the relaxed-ring generate more messages, but they are only triggered when they are needed.

5. Future Work

Apart from the simulator used for the validation, we have tested the Relaxed-Ring using a real implementation running distributed processes on small networks. We are currently testing our implementation on PlanetLab [1] to address more aggressive environments, and where we expect to report more about on failure recovery. 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. We also plan to build a replicated transactional distributed hash table based on a modified Paxos consensus algorithm [12].

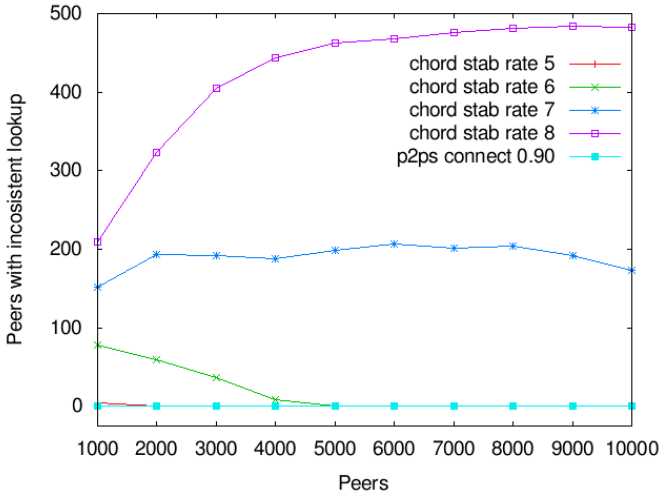

Fig. 10. Amount of peers with overlapping ranges of responsibilities, introducing lookup inconsistencies, on Chord networks under different stabilization rates for different network sizes. Comparison with the Relaxed-Ring (p2ps) with a bad connectivity. The stabilization rate represent the amount of peers joining/leaving the network between every stabilization round. The value of zero in the Y-axis has been raised in order to spot the curve of the Relaxed-Ring and Chord with a very frequent stabilization rate equal to 5.

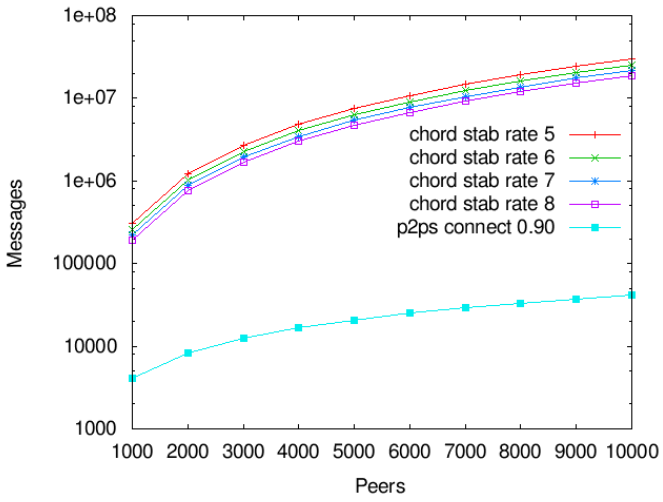

Fig. 11. Load of messages in Chord due to periodic stabilization, compared to the load of the Relaxed-Ring maintenance with bad connectivity. Y-axis presented in logarithmic scale.

6. Conclusion

In this paper we have presented a novel Relaxed-Ring topology for fault-tolerant and self-organizing peer-to-peer networks. The topology is derived from the simpli-

fication 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 they help the system to work in realistic scenarios. The topology adds some complexity to the routing algorithm, but it does not degrade the complexity of its performance. We consider this issue a small drawback in comparison to the gain in fault tolerance and cost-efficiency in ring maintenance.

The topology makes feasible the integration of peers with very poor connectivity. Having a connection to a successor is sufficient to be part of the network. Leaving the network can be done instantaneously without having to follow a departure protocol, because the failure-recovery mechanism will deal with the missing node. 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.

Acknowledgements

The authors would like to thank the *distoz* group at Université catholique de Louvain and S. González for comments on this work. This research is mainly funded by SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] PlanetLab. *http://www.planet-lab.org*, 2008.

[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 344, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Eric 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.

[4] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[5] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://p2ps.info.ucl.ac.be*, 2008.

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

[7] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity, 2003.

[8] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.

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

[10] Boris Mejías. CiNiSMO: Concurrent Network Simulator in Mozart-Oz, Université catholique de Louvain, Belgium. *http://p2ps.info.ucl.ac.be/cinismo*, 2008.

[11] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, November 2007.

[12] Monika Moser and Seif Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[13] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 2008.

[14] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08)*. ACM, June 2008.

[15] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.