A Relaxed-Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks[*]

Boris Mejías and Peter Van Roy
Université catholique de Louvain,
Louvain-La-Neuve, Belgium
{bmc|pvr}@info.ucl.ac.be

Abstract

There is no doubt about the increase in popularity of decentralised systems over the classical client-server architecture in distributed applications. These systems are developed mainly as peer-to-peer networks where it is possible to observe many strategies to organise the peers. The most popular one for structured networks is the ring topology. Despite many advantages offered by this topology, the maintenance of the ring is very costly, being difficult to guarantee lookup consistency and fault-tolerance all the time. By increasing self-management in the system we are able to deal with these issues. We model ring maintenance as a self-organising and self-healing system using feedback loops. As a result, we introduce a novel relaxed-ring topology that is able to provide fault-tolerance with realistic assumptions concerning failure detection. Limitations related to failure handling are clearly 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.

Index Terms — Decentralised systems, Peer-to-peer, Fault-tolerance, Self-management, Feedback-loops

1. Introduction

Decentralised applications has rapidly increased their popularity in the last years due to several factors and motivations. The increase of Internet bandwidth with a sufficient reliability is already an important element. The fact that home computers have augmented their computing power has decreased the dependency on big servers, because clients are powerful enough to play the role of a server for several tasks. These factors have allowed the introduction of peer-to-peer networks. Such networks have reduced the problem of traffic congestion and single point of failure as in the client-server architecture, making decentralised applications popular.

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. Other wished properties such as efficient routing, scalability and full reachability, made that randomly connected peer-to-peer networks moved towards structured overlay networks. Many of these structured networks implements a Distribute Hash Table (DHT). Among many of them - Pastry [16], Tapestry [20], Kademlia [12], HyperCup [17], P-Grid [1] - we focus on Chord [18], because it is quite representative and it introduces a ring topology that has influenced many other networks.

In Chord, peers are organised in a ring, having a set of pointers to efficiently find any other peer in the network. The resources of the system are distributed among the peers where each one is responsible for a set of them. Performing a lookup for a resource must result in a consistent answer, finding the right responsible. To add or remove a peer from the network, the peer only needs to synchronise with its direct neighbours. The network is self-organising, meaning that peers will organise themselves in the ring topology without needing manual reconfiguration.

Despite the self-organising nature of the ring architecture, its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents temporary inconsistency when several peers join the network concurrently. This problem occurs even in fault-free scenarios. To fix these inconsistencies, a stabilisation protocol must be run periodically. The system must also deal with peers gently leaving the network, which can occur massively and concurrent to other joining events. The most challenging issue though, is failure handling, where peers just leave the network breaking the ring without following any protocol.

As we can see, ironically, the advantages of decentralised systems with respect to the classical client-server architecture, have the drawback of higher complexity due to the lack

[*]This work is being funded and developed in the context of projects EVERGROW (contract number:001935) and SELFMAN (contract number: 034084).

of a single point of control and synchronisation. Increasing self-management of decentralised systems can help us to reduce this new complexity. By self-management we mean the ability of a system to maintain its functionality despite changes in its environment. The system constantly monitor itself triggering corrective actions when the current state deviates from the desired one. In order to achieve self-management, the use of feedback loops in the design of the system appears as a straight forward approach.

We use feedback loops to model the ring-maintenance of our peer-to-peer system, called P2PS [13], which also uses a ring topology. As a result of this new design, we introduce a novel *relaxed-ring* topology that simplifies the "join" algorithm and greatly improves failure recovery. Having the ability of handling failures, there is no need for a "leave" algorithm, because this case is already covered by failure recovery.

The main contribution of this work is the design of a peer-to-peer network as a self-managing system, introducing a relaxed-ring topology that is able to provide fault-tolerance with realistic assumptions concerning failure detection. The use of feedback loops for modelling the system can be reused not only in other decentralised systems, but also in software design in general.

Section 2 gives a more detailed introduction to peer-to-peer networks using ring topology, describing some existing solutions for ring maintenance. Section 3 briefly introduces feedback loops for self-managing systems and how they can be applied to software design. The result of applying feedback loops to the ring maintenance is given in section 4 with a detailed description of the *relaxed-ring*. After a deep analysis of failure handling, the paper provides conclusions for this work.

2. Peer-to-Peer Rings

Peer-to-peer networks appear as the evident framework for working with decentralised systems. Looking at the history of peer-to-peer systems, we find Napster[15] as the icon of the first generation. Napster uses a hybrid architecture with a centralised directory storing the location of the resources of the systems. Thus, the client-server strategy was needed in order to find other peers.

A second generation characterised by Gnutella [8] and FreeNet [6] removed the servers from the topology becoming the first real peer-to-peer network. Peers build an overlay network on top of the Internet, being able to route with its own topology. No structure is used for the network because peers are connected randomly to other peers. Therefore, no strong guarantees can be provided with respect to reachability, availability and time to find items. Unfortunately, these kind of network have limited scalability and induce a huge amount of traffic [11].

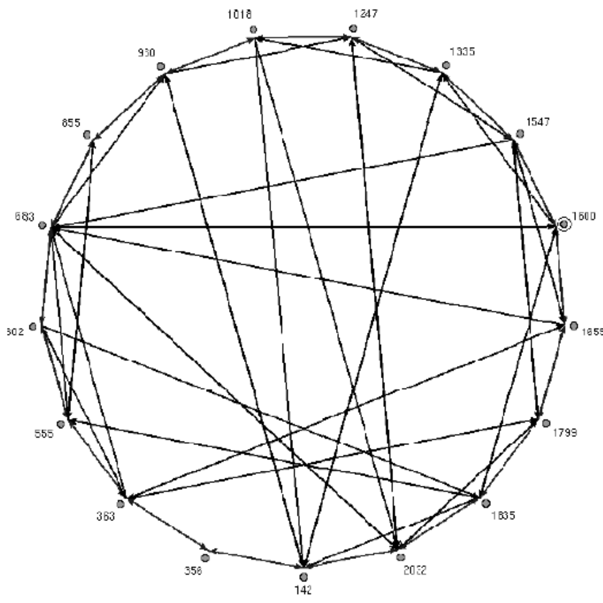

Figure 1. Structured overlay network using ring topology

Structured overlay networks - see introduction for references - appear as the third generation of peer-to-peer systems, claiming self-organisation of the network with fault-tolerance in addition to the guarantees that cannot be found in the second generation.

Figure 1 depicts a structured overlay network using ring topology and providing a DHT with election of fingers based on the Tango [3] algorithm. This structure was first introduced by Chord [18]. Every peer is identified with a hash key, and it is connected to a successor and a predecessor respecting the order of the keys in clockwise direction. The DHT is used for storing and finding items in the network using basically two operations: $put(key, value)$ to store a value with a certain key, and $get(key)$ to recover the value. Every peer is responsible for all keys between its predecessor's identifier and itself, excluding the predecessor to avoid overlapping. When lookup for a key is triggered from any point of the ring, consistency must be guaranteed. We define this as follows:

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.*

As we mentioned already, ring maintenance is costly and it is not trivial to guarantee correctness. This is because the state of the ring is updated concurrently without having any centralised point of synchronisation. Chord's algorithms for ring maintenance, handling *joins* and *leaves*, present well known problems of temporary inconsistency, where more that one peer appears to be the responsible for a key. For this reason, Chord needs to trigger periodic stabilisation in order to fix the inconsistencies. Existing analyses [7] conclude that the problem comes from the fact that joins and leaves are not atomic operations. We also raise the issue that these operations always need the synchronisation of three peers, which is hard to guarantee with asynchronous communication, which is inherent to distributed programming.

Existing solutions [9, 10] introduce locks in the algorithms in order to provide atomicity to *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 [7], simplifying the locking mechanism and proving correctness of the algorithms in absent of failures. 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 locked peers must be suspended in presence of join or leave in order to guarantee consistency. Leaving peers are not allowed to leave the network until they are granted with the relevant locks. Given that, peers crashing can be seen as peers just leaving 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 situation is worse when the peer holding the relevant lock is the one that crashes. Under this considerations, we can observe that locks in distributed systems can hardly present an efficient fault-tolerant solution.

3. Feedback Loops

Taken from system theory, *feedback loops* can be observed not only in existing automated systems, but also in self-managing systems in nature. Several examples of this can be found in [19], where feedback loops are introduced as a designing model for self-managing software. The loop consists out of three main concurrent components interacting with the subsystem. There is at least one agent in charge of monitoring the subsystem, passing the monitored information to a another component in charge of deciding a corrective action if needed. An actuating agent is used in order to perform this action in the subsystem. Figure 2 depicts the interaction of these three concurrent components in a feedback loop. These three components together with the subsystem forms the entire system. It has similar properties to *PID-controllers*, with the difference that the evolution of a running software application is measured discretely.

The goal of the feedback loop is to keep a global property of the system stable. In the simplest cases, this property is represented by the value of a parameter. This parameter is constantly monitored. When a perturbation is detected,

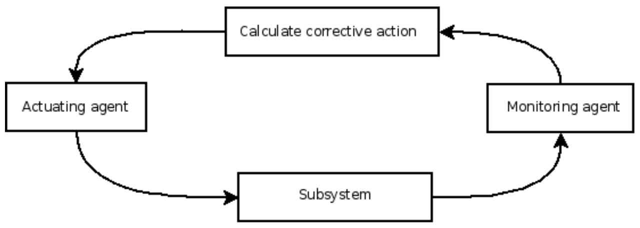

Figure 2. Basic structure of a feedback loop (taken from [19])

a corrective action is triggered. A negative feedback will make the system reacts in the opposite direction to the perturbation. Positive feedback increases the perturbation.

Taking an air-conditioning as example, we can see the *room* where the system is installed as the subsystem. A thermometer constantly *monitors the temperature* in the room giving this information to a *thermostat*. The thermostat is the component in charge of computing the correcting action. If the monitored temperature is higher than the wished temperature, the thermostat will decide to *run the air-conditioning* to cool it down. That action corresponds to the actuating agent.

Since every component executes concurrently, the model fits very well for modelling distributed systems. There are many alternatives for implementing every component and the way they interact. They can represent active objects, actors, functions, etc. Depending on the chosen paradigm, the communication between components can be done for instance by message passing or event-based communication. The communication may also be triggered by pushing or pulling, resulting on eager or lazy execution.

Independent of the strategy used for communication, it is important to consider asynchronous communication as the default when distributed systems are being modelled.

As a rule for using feedback loops in the design of a system, actuators and monitors appear as verbs, while the subsystem and the computing component appear as substantives, as in the air-conditioning example. The reason why it is not like this in Figure 2, is because that is a description of the model, and not the model applied to a system.

4. Self-Organising and Self-Healing Relaxed-Ring

Section 2 described the problem of guaranteeing consistent lookup while multiple joins, leaves and failures occur in a peer-to-peer network using ring architecture. As a solution to this problem we design a novel topology based on a relaxed-ring. This topology also allows as to provide failure recovery using imperfect failure detectors and handling broken links which are often ignored. The relaxed-ring topol-

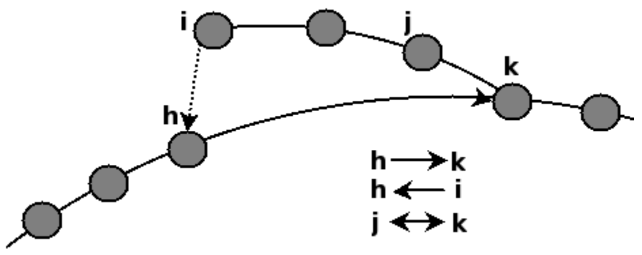

Figure 3. Branch created due to connection problems between peers p and q

ogy is part of the new version of P2PS [5], implemented with Mozart-Oz programming system [14].

During this section we will use the terms *peer* and *node* indistinctly to refer to an independent process running with its own address space, i.e., a network node. We also use the term *pointer* as a network reference to a node. The terms *key* and *identifier* represent keys from the DHT, and they are used to identify peers.

The algorithms of the relaxed-ring are designed using feedback loops, and the description of their implementation is given using event-driven notation. 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. The ring provides a DHT with key-distribution formed by integers from 0 to N growing clockwise.

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 by 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 peers are joining the network. Another relevant difference with the mentioned related work, is that we do not rely on graceful leaving of peers, because anyway, we have to deal with leaves due to network and node failures. For efficiency, a graceful leaving peer could communicate its departure to its neighbours in order to avoid the timeout in failure detection.

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 *core ring*. When there are no branches, and every peer is connected bidirectionally with its successor and predecessor, then we have a *"perfect ring"*.

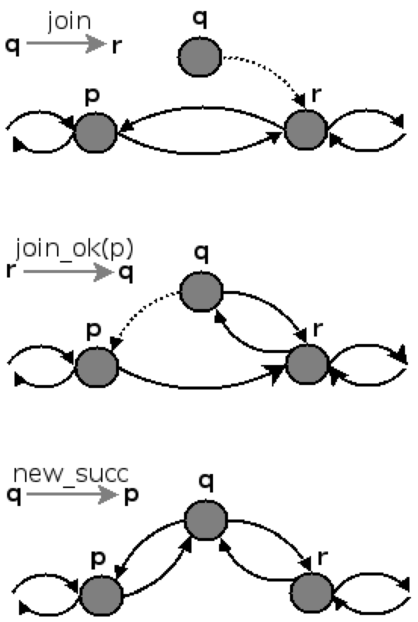

Figure 4. Messages and pointers update during a join

Figure 3 shows a fraction of a relaxed ring where peer t is the root of a branch, and where the connection between peers p and q is broken. We say that p and t belongs to the *core ring*, and that q, r and s are part of a branch.

4.1. The join algorithm

Thinking about the peer-to-peer network as self-managing system, the network is the subsystem we want to monitor, because we want it to keep is functionality despite the changes that can occur. The structure of the ring is the global property that needs to be kept stable. New peers joining, and current peers leaving or failing represent perturbations to the ring structure. Therefore, these events must be monitored.

Messages sent during the process of joining, and the update of the predecessor and successor pointers are shown in figure 4. In the example, node q wants to join the network

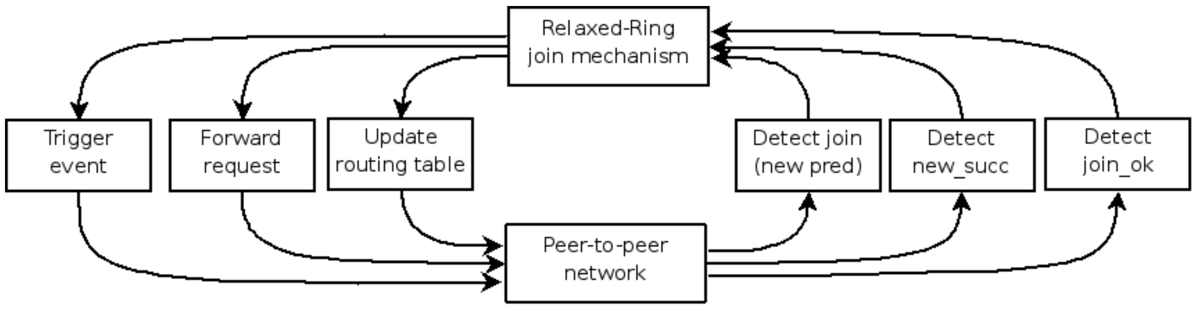

Figure 5. Join algorithm as a feedback loop

having r as successor candidate. Peer r is a good candidate because it is the responsible for key q. Node q send a join request to r. Whereas event $join$ triggered by peer q is a perturbation, event $join\_ok$ is a correcting action providing negative feedback. It is negative because it is an action that goes in the opposite direction of the perturbation. After $join\_ok$ is triggered, a branch is created. Then, a second correcting action is needed to entirely close the ring. This action is represented by the event new_succ sent from peer q to p.

Figure 5 describes the feedback loop that keeps the structure of the relaxed-ring stable. The monitoring agents are in charge of detecting perturbations in the network. Correcting actuators can be seen as three different actions: update routing table (successor and predecessor), trigger event (correcting ones) and forward request (in case a peer wants to join in the wrong place). The routing table does not only include predecessor and successor. It also includes fingers for efficient routing and resilient sets for failure recovery.

Every peer is independently monitoring the network, and the correcting action performing the ring maintenance is running concurrently in every peer. Every event triggered by a peer is monitored by the destination peer, unless there is a failure in the communication. In that case, a *crash* event will be triggered and treated by the failure recovery mechanism.

Algorithm 1 describes one implementation of the feedback loop. Every event is handled by the computing component running in every peer. This component decides which correction has to be performed. In event $join$, the messages $goto$ and try_later represent the forwarding of the request. The request can be accepted when the joining peer is a $betterPredecessor$. This is the case when $q \in (pred, self]$. As part of the joining process, there is an update of the routing table. This update is done explicitly by assigning the corresponding pointer $pred$ and the $pred\_list$.

Operator **send** is a reliable point-to-point send. If the receiver presents a failure before the message arrives, the sender is notified.

Triggering correcting events is represented by the message $join\_ok$, which will be monitored by the joining peer. Handling event $join\_ok$ also shows how the routing table is updated by assigning pointer $succ$ and set $succ\_list$. A second correcting event is triggered: new_succ. The set named $succ\_list$ is used later for failure recovery. This set represents the list of peers that follows after the current successor. This peers can be contacted in order to fix the ring when the successor is suspected of having a failure.

Algorithm 1 Join step 1 - adding a new node

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

17: upon event ⟨ $join\_ok$ | p, r, sl ⟩ do
18:     succ := r
19:     succlist := {r} ∪ sl
20:     if ($pred = nil$) ∨ ($p \in (pred, self)$) then
21:         pred := p
22:         send ⟨ $new\_succ$ | self, succ, succlist ⟩ to $p$
23:     end if
24: end event
```

Note that the algorithm is divided into two steps. Like this, we do not need the synchronisation of three peers performing an atomic operation. Instead, two correcting actions are triggered in order to fix the perturbation. Algorithm 2 describes the implementation of the second action where the ring is closed again. This is achieved by updating pointer $succ$ and set $succ\_list$, which are part of the routing table. A notification event $join\_ack$ is triggered to improve the knowledge of the system about its global state, but it is not strictly needed.

Algorithm 2 Join step 2 - Closing the ring

```
1:  upon event ⟨ $new\_succ$ | q, olds, sl ⟩ do
2:      if ($succ = olds$) then
3:          oldsucc := succ
4:          succ := q
5:          succlist := {q} ∪ 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 ($op \in predlist$) then
12:         predlist := predlist \ {op}
13:     end if
14: end event
```

It is important to signalise that the routing algorithm of Chord or DKS [2] cannot be used in the relaxed-ring. The algorithm would create cycles due to the introduction of branches in the ring topology. The routing algorithm of the relaxed-ring works as follows. A peer i must choose the closest peer j to the key k from its routing table. The routing table also consider predecessors for routing. The distance function between two keys is given by $d(k, j) = (j - k) modN$, where N is the highest value of the key domain. If the relaxed-ring is a perfect ring, the routing algorithm behaves like in Chord and DKS, converging to the responsible of the key in $log_f(N)$ hops, where f is the amount of fingers per peer. In presence of branches, it converges in $log_f(N)$ to the root of the branch where the responsible is connected. In the worse case, the routing algorithm still have to traverse the branch. This gives a complexity of $log_f(N) + B$, where B is the size of the branch. If there is good connectivity, the size of branches tend to zero, behaving like Chord. If the connectivity is bad, routing will take a bit longer when a branch is involved, but the network will work correctly. This is an advantage with respect to Chord or DKS, because they will not behave correctly if connectivity is bad, because they relay on perfect predecessor-successor relationship.

Given the join and routing algorithms, the relaxed-ring guarantees consistent lookup at any time in presence of multiple joining peers. To prove this guarantee, let us assume the contrary. Then, 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.

4.2. Resilient information

During the join algorithm 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 4.3 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

Algorithm 3 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. layers of P2PS.

Algorithm 3 Update of successor list

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

4.3. Failure recovery

Instead of designing a costly protocol for peers leaving the network, leaving peers are treated as network nodes having a failure. Like this, solving problem of failure recovery will also solve the issue of leaving the network.

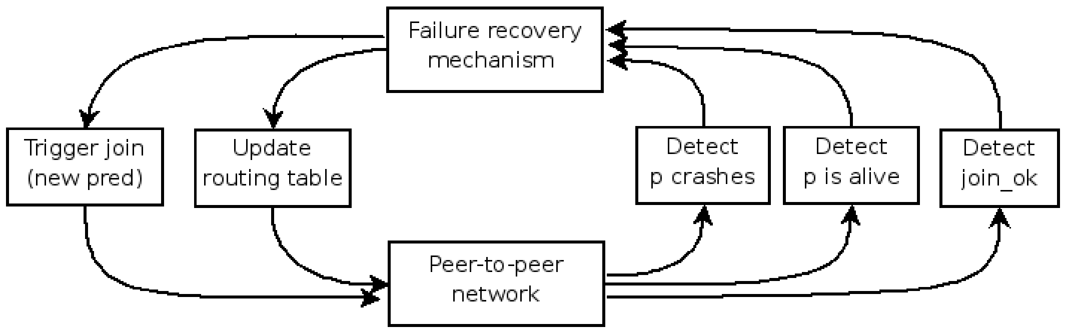

Figure 6. Failure recovery as a feedback loop

Observing the relaxed-ring as a self-managing system, we identify that the crash of a peer also introduces perturbations to the structure of the ring. Therefore, crashes must be monitored. In order to provide a realistic solution, *perfect failure detectors* cannot be assumed. Perfect failure detectors are strongly complete and strongly accurate. Being complete means that every crashed node is detected. Being accurate means that a node being suspected of failure is effectively in failure. In reality, broken links and nodes with slow network connection are very often, generating a considerable amount of false suspicions. Because of this, not only crashed events must be monitored, but also “I am alive” messages. When these two events are appear as perturbations, the network must update routing tables and trigger correcting events.

Algorithm 4 Failure recovery

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

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

In the relaxed-ring architecture we reuse the $join$ event as correcting agent for stabilising the relaxed-ring. If the network become stable, the $join\_ok$ event will be monitored. This negative feedback can be observed in figure 6.

Algorithm 4 describes an implementation of the feedback loop. If a failure is detected, the $crash$ event is triggered. The detected node is removed from the 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. A 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. Note that as every crashed peer is immediately removed from the resilient sets, $getFirst$ always returns 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. This action generates an interaction between the two feedback loops. If the successor candidate also fails, a new candidate will be chosen. This is verified with 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 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 peer recovers from a temporary failure, the *alive* event is triggered. This can be implemented by using watchers or using a fault stream attached to the distributed entities [4]. To handle the *alive* event 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.

Having now the knowledge of the $crashed$ set, algorithm 5 gives a complete definition of the function $betterPredecessor$ used in algorithm 1. Since the $join$ event is used both for a regular join and for failure recovery, the function will decide if a predecessor candidate is better than the current one if it belongs to its range of responsibil-

Algorithm 5 Verifying predecessor candidate

```
function betterPredecessor($q$) is
    if ($q \in (pred, self)$) then
        return ($true$)
    else
        return ($pred \in crashed$)
    end if
end function
```

ity, or if the current $pred$ is detected as a faulty peer.

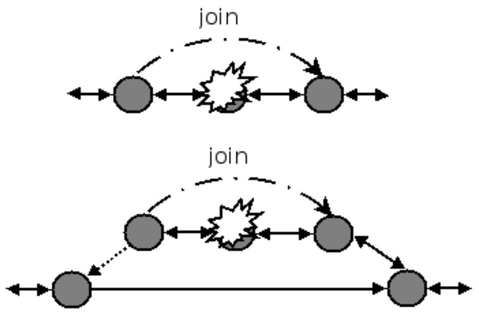

Figure 7. Failure recovery triggered in the ring and in a branch

Knowing the recovery mechanism of the relaxed-ring, let us come back to our joining example of figure 4 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.

Figure 7 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.

Figure 8 shows two simultaneous crashes together with a new peer concurrently joining the network. 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 $join$

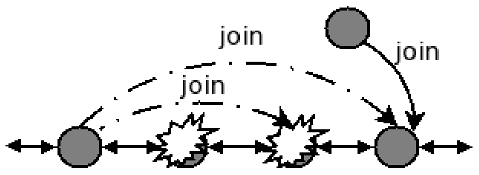

Figure 8. Simultaneous crashes together with a join event

message arrive, the recovering peer will be forwarded to the new peer. The contact of these two peers will finally fix the ring and removing the branch.

There are failures more difficult to handle than the ones we have already analysed. Figure 9 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. This will happen when the link is recover from the failure, and the *alive* event is monitored.

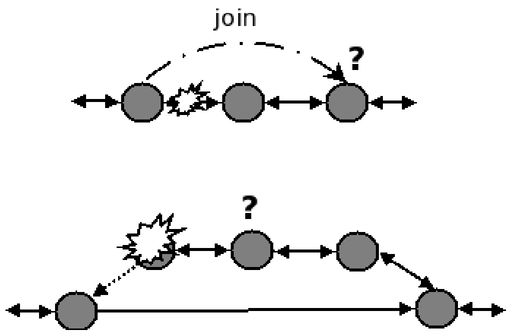

Figure 9. Broken link and failure of the tail of branch

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, which will be discussed in section 4.5.

With respect to failure handling, the relaxed-ring guarantees that simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition. To prove this guarantee, we must consider that every failure of a peer 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 fail-

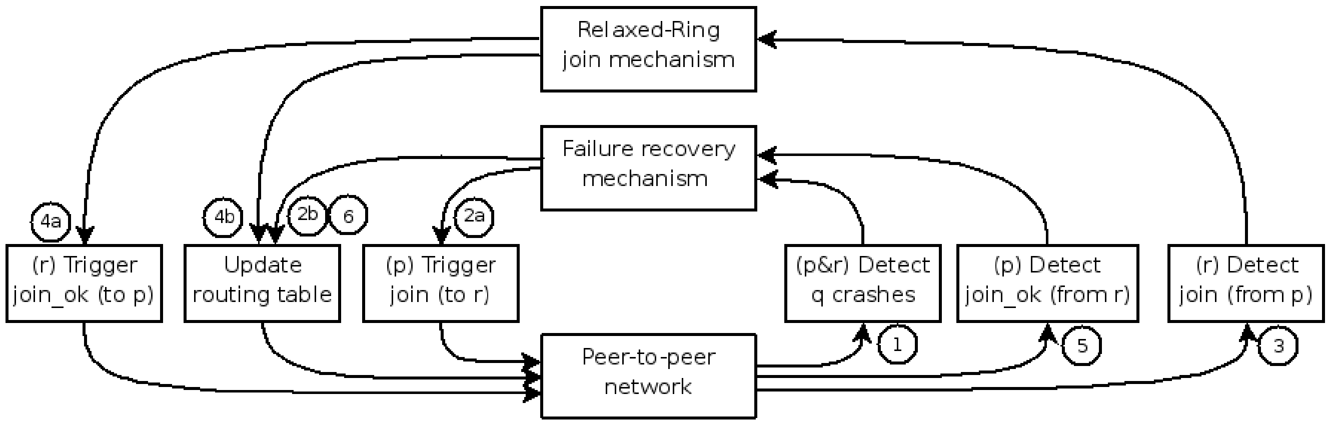

Figure 10. Peers p and r detect failure of q, fixing the ring with an interaction of feedback loops

ure 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.

4.4. Combining feedback loops

The interaction between feedback loops is an interesting issue to analyse because big systems are expected to be designed as a combination of several loops. Let us consider a particular section of the ring having peers p, q and r connected through successor and predecessors pointers. Figure 10 describes how the ring is perturbed and stabilised in the presence of a failure of peer q. Only relevant monitored and actuating actions are included in the figure to avoid a bigger and verbose diagram.

Initially, the crash of peer q is detected by peers p and r (1). Both peers will update their routing tables removing q from the set of valid peers (2b). But, since p is q's predecessor, only p will trigger the correcting event $join$ (2a). This first iteration corresponds to a loop from the failure recovery mechanism. The $join$ event will be monitored by peer r (3), starting an iteration in the join maintenance loop. The correcting action $join\_ok$ will be triggered (4a) together with the corresponding update of the routing table (4b). Then, the event $join\_ok$ will be monitored (5) by the failure recovery component in order to perform the correspondent update of the routing table (6). Since the $join\_ok$ event is also detected by the join loop, both loops will consider the network stable again.

4.5. Limitations

Figure 11 depicts a temporary network partition that can occur in the relaxed-ring topology. Previously, we have analysed cases where 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. This situation introduces a temporary inconsistency. This limitation is not unique to the relaxed-ring topology. It is related to the proof given by Ghodsi in [7], where it is not possible to provide at the same time consistency, availability and partition-tolerance in presence of network partitioning. The limitation of the particular case of the relaxed-ring is well defined in the following theorem.

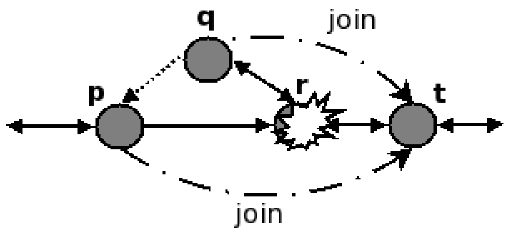

Figure 11. The failure of the root of a branch triggers two recovery events

Theorem 4.1 *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 1 *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 4.1 clearly states the limitation of branches in the system. This helps developers to identify scenarios where special failure recovery must be taken into account. Since the problem is related to network partitioning, there seems to be no easy solution for it. An advantage of the relaxed-ring 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.

5. Conclusions

Decentralised systems in the form of peer-to-peer networks presents many advantages over the classical client-server architecture. Even though, the complexity of a decentralised system is higher, requiring the increase of self-management. In this paper we show how feedback-loops, taken from existing self-managing systems, can be applied in the design of a peer-to-peer network. The result is a novel relaxed-ring topology for fault-tolerant and self-organising networks. The system is able to monitor and correct itself, keeping the ring structure stable despite the changes due to regular operations of due to network and node failures.

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 ability to handle failures removes the need for a *leave* algorithm, because it is just a special case in the failure recovery mechanism.

Related work is discussed along the paper, but it is specially analysed in section 2. The guarantees and limitations of the relaxed-ring of P2PS are clearly identified and formally stated in section 4. These specifications provide helpful indications to developers in order to build fault-tolerant applications on top of this structured overlay network.

References

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. *Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Lecture Notes in Computer Science*, 2172:179–194, 2001.

[2] L. O. Alima, S. El-Ansary, P. Brand, and S. 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] B. Carton and V. Mesaros. Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 1060–1067. Springer, 2004.

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

[5] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://gforge.info.ucl.ac.be/projects/p2ps*, 2007.

[6] FreeNet. http://freenet.sourceforge.net, 2003.

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

[8] Gnutella. http://gnutella.com, 2003.

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

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

[11] E. P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[12] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[13] V. Mesaros, B. Carton, and P. Van Roy. P2PS: Peer-to-peer development platform for mozart. In P. Van Roy, editor, *MOZ*, volume 3389 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 2004.

[14] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 2007.

[15] Napster. Open source napster server, 2002.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2001.

[17] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. Hypercup – hypercubes, ontologies and efficient search on p2p networks, 2002.

[18] 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.

[19] P. Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, September 2006.

[20] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.