IMPROVING THE PEER-TO-PEER RING FOR BUILDING FAULT-TOLERANT GRIDS

Boris Mejías and Donatien Grolaux and Peter Van Roy
Université catholique de Louvain, Belgium *
{bmc|ned|pvr}@info.ucl.ac.be

Abstract

Peer-to-peer networks are gaining popularity in order to build Grid systems. Among different approaches, structured overlay networks using ring topology are the most preferred ones. However, one of the main problems of peer-to-peer rings is to guarantee lookup consistency in presence of multiple joins, leaves and failures nodes. Since lookup consistency and fault-tolerance are crucial properties for building Grids or any application, these issues cannot be avoided. We introduce a novel relaxed-ring architecture for fault-tolerant and cost-efficient ring maintenance. Limitations related to failure handling are formally identified, providing strong guarantees to develop applications on top of the relaxed-ring architecture. Besides permanent failures, the paper analyses temporary failures and broken links, which are often ignored.

Keywords: Peer-to-peer, relaxed-ring, fault-tolerance, lookup consistency, ring maintenance.

1. Introduction

Classical approaches for building Grid systems using resource and service discovery are mainly centralised or hierarchical. Since centralised networks present the weakness of having single point of failure, peer-to-peer networks are gaining popularity as an alternative decentralised choice. Building decentralised applications requires several guarantees from the underlay peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Structured overlay network providing a Distributed Hash Table (DHT) using Chord-like ring topology [10]are a popular choice to solve the requirements of efficient routing, lookup consistency and accessibility of all resources. But all these properties are compromised in

*This research is mainly funded by EVERGROW (contract number:001935) and SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

presence of failure or high churn (multiple peers joining or leaving in very short time).

The benefits of using peer-to-peer systems have been already stated in previous CoreGRID results [12, 11], but the problems related to fault-tolerance has not been deeply addressed. A high level approach is proposed in [2], where the failure detection and self-organisation of the network is entirely delegated to the peer-to-peer system. Since this work addresses these issues precisely at the low level, it can be seen as a complementary result.

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 with massive peers joining the network, even in fault-free systems. A stabilisation protocol must be run periodically to fix these inconsistencies. Existing analyses conclude that the problem comes from the fact that joins and leaves are not atomic operations, and they always need the synchronisation of three peers. Synchronising three peers is hard to guarantee with asynchronous communication, but this is inherent to distributed programming.

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

We have developed an algorithm that only needs the agreement of two nodes at each stage, which is easier to guarantee given point-to-point communication. This decision leads us to a relaxed-ring topology, simplifying the joining algorithm and becoming fault tolerant to permanent or temporary failures of nodes, and also to broken links, which are often ignored by existing approaches.

The following section describes the relaxed-ring architecture and its guarantees. We continue with further analysis of the topology and its fault tolerant behaviour, ending with conclusions.

2. P2PS's relaxed-ring

The relaxed-ring topology is part of the new version of P2PS [4], which is designed as a modular architecture based on tiers. The whole system is implemented using the Mozart-Oz programming system [9], where the lowest level tier implements point-to-point communication between peers. Some layer upper to this one, we implement the maintenance of the relaxed-ring topology, which is the focus of this paper. This layer can correctly route lookup requests providing consistency. Other layers built on top of this one are in charge of providing efficient routing, reliable message sending, broadcast/multicast primitives and naming services. All these layers provide efficient support for building decentralised systems such as grid based on services architectures like P2PKit [6].

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

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

2.1 The relaxed-ring

As we previously mentioned, one of the problem we have observed in existing ring maintenance algorithms is the need for an agreement between three peers to perform a join/leave action. We provide an algorithm where every step only needs the agreement of two peers, which is guaranteed with a point-to-point communication. In the specific case of a join, instead of having one step involving 3 peers, we have two steps involving 2 peers. The lookup consistency is guaranteed between every step and therefore, the network can still answer lookup requests while simultaneous nodes are joining the network. Another relevant difference is that we do not rely on graceful leaving of peers, because anyway, we have to deal with leaves due to network failures.

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

from the *"perfect ring"*. Figure 1 shows a fraction of a relaxed ring where peer k is the root of a branch, and where the connection between peers h and i is broken.

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

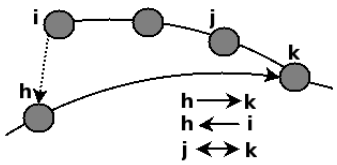

Figure 1. The relaxed-ring architecture

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

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

When a new peer wants to join the ring, first, it gets its own identifier from a random key-generator. At this starting point, the node does not have a successor ($succ$), then, it does not belong to any ring, and it does not know its predecessor ($pred$), so obviously, it does not have responsibilities. Having an access point, that can be any peer of the ring, the new peer triggers a lookup request for its own key in order to find its best successor candidate. This is quite usual procedure for several Chord-alike systems. When the responsible of the key contacts the new peer, it begins the join algorithm that will be discussed in the next section.

2.2 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. When peer r replies the lookup request to q, and q send the $join$ message to r triggering the joining process.

The first step is described in algorithm 1, 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.

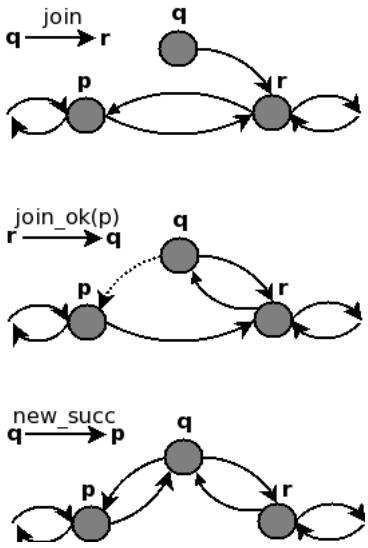

Figure 2. The join algorithm.

If it is possible to perform the join, peer r verifies that peer q is a better predecessor. Function $betterPredecessor$ just checks if the key of the joining peer is in the range of responsibility of the current peer in the case of a regular join. 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 events $reply\_lookup$ and $join$. In that case, q will be redirected to the corresponding peer with the $goto$ message, eventually converging to the responsible of its key.

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

Note that before updating the predecessor pointer, peer q must verify that its predecessor pointer is nil, or that it belongs to its range of responsibility. This second condition is only used in case of failure recovery and it will be described in section 3. 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 2. 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

Algorithm 1 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
```

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.

Algorithm 2 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
```

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. Note that message $upd_succlist$ is for resilient purposes. It updates the list of successors that will be used for the recovery of a failure detected in the successor. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

THEOREM 2.1 *The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.*

PROOF 1 *Let us assume the contrary. There are two peers p and q responsible for key k. In order to have this situation, p and q must have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$. The join algorithm updates the predecessor pointer upon events $join$ and $join\_ok$. In the event $join$, the predecessor is set to a new joining peer j. This means that no other peer was having j as predecessor because it is a new peer. Therefore, this update does not introduce any inconsistency. Upon event $join\_ok$, the joining peer j initiates its responsibility having a member*

of the ring as predecessor, say i. The only other peer that had i as predecessor before is the successor of j, say p, which is the peer that triggered the $join\_ok$ event. This message is sent only after p has updated its predecessor pointer to j, and thus, modifying its responsibility from $(i, p]$ to $(j, p]$, which does not overlap with j's responsibility $(i, j]$. Therefore, it is impossible that two peers has the same predecessor.

3. Failure Recovery

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

Every node monitors the communication with every peer it is connected to. If a failure is detected, the $crash$ event is triggered as it is described in algorithm 3. The detected node is removed from the resilient sets $succlist$ and $predlist$, and added to a $crashed$ set. If the detected peer is the successor, the recovery mechanism is triggered. The $succ$ pointer is set to nil to avoid other peers joining while recovering from the failure, and the successor candidate is taken from the successors list. The function $getFirst$ returns the peer with the first key found clockwise, and removes it from the set. It returns nil if the set is empty. Function $getLast$ is analogue. Note that as every crashed peer is immediately removed from the resilient sets, these two functions always return a peer that appears to be alive at this stage. The successor candidate is contacted using the $join$ message, triggering the same algorithm as for joining. If the successor candidate also fails, a new candidate will be chosen. This is verified in the if condition.

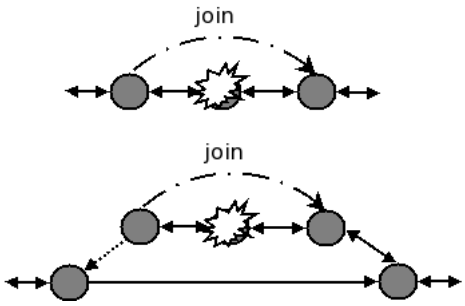

Figure 3. Simple crashes.

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.

The $alive$ event is triggered when a link recovers from a temporary failure. This can be implemented by using watchers or a fault stream per distributed entity [3]. If the peer is alive, it is enough to remove it from the $crashed$ set. This will terminate any pending recovery algorithm.

Algorithm 3 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:     else if (p == pred) then
10:        if (predlist ≠ ∅) then
11:            pred_candidate := getLast(predlist)
12:        end if
13:    end if
14: end event

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

Figure 3 shows the recovery mechanism triggered by a peer when it detects that its successor has a failure. The figure depicts two equivalent situations. Using the $crashed$ set, function $betterPredecessor$ can check fault status. 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.

Knowing the recovery mechanism of the relaxed-ring, let us come back to our joining example and check what happens in cases of failures. If q crashes after the event $join$, peer r still has p in its $predlist$ for recovery. If q crashes after sending new_succ to p, p still has r in its $succlist$ for recovery. If p crashes before event new_succ, p's predecessor will contact r for recovery, and r will inform this peer about q. If r crashes before new_succ, peers p and q will contact simultaneously r's successor for recovery. If q arrives first, everything is in order with respect to the ranges. If p arrives first, there will be

two responsible for the ranges $(p, q]$, but one of them, q, is not known by any other peer in the network, and it fact, it does not have a successor, and then, it does not belong to the ring. Then, no inconsistency is introduced in any case of failure.

Since failures are not detected by all peers at the same time, redirection during recovery of failures may end up in a faulty node. Then, the $goto$ event must be modified such that 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.

Cases hard to handle are broken links and crashes at the tail of a branch. In the case of the broken link (inaccuracy), the failure recovery mechanism is triggered, but the successor of the suspected node will not accept the join message. The described algorithm will eventually recover from this situation when the failure detector reaches accuracy. In the case of the crash of the node at the tail of a branch, there is no predecessor to trigger the recovery mechanism. In this case, the successor could use one of its nodes in the predecessor list to trigger recovery, but that could introduce inconsistencies if the suspected node has not really failed. If the tail of the branch has not really failed but it has a broken link with its successor, then, it becomes temporary isolated and unreachable to the rest of the network. Having unreachable nodes means that we are in presence of network partitioning. The following theorem describes the guarantees of the relaxed-ring in case of temporary failures with no network partitioning.

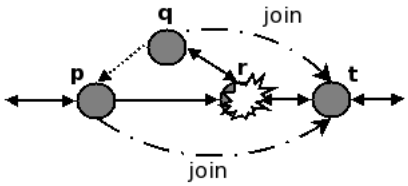

Figure 4. The failure of the root of a branch triggers two recovery events

THEOREM 3.1 *Simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition.*

PROOF 2 *Every failure of a node is eventually detected by its successor, predecessor and other peers in the ring having a connection with the faulty node. The successor and other peers register the failure in the $crashed$ set, and remove the faulty peer from the resilient sets $predlist$ and $succlist$, but they do not trigger any recovery mechanism. Only the predecessor triggers failure recovery when the failure of its successor is detected, contacting only one peer from the successor list at the time. Then, there is only one possible candidate to replace each faulty peer, and then, it is impossible to have two responsible for the same range of keys.*

With respect to network partitions, there are two important cases we want to analyse. The crash of a branch's root, and the isolation of a set of nodes from

the rest of the ring. The isolation problem can occur in any system using ring topology, and it can involve consecutive peers or peers distributed all over the ring. Network partitioning introducing temporary uncertainty has been proved by Ghodsi [5], and it is related to the proof provided in [1]about limitations of web services in presence of network partitioning.

Figure 4 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 3.1 is based on the fact that per every failure detected, there is only one peer that triggers the recovery mechanism. In the case of the failure of the root of a branch, peer r in the example, there are two recovery messages triggered by peers p and q. If message from peer q arrives first to peer t, the algorithm handle the situation without problems. If message from peer p arrives first, the branch will be temporary isolated, behaving as a network partition introducing a temporary inconsistency. This limitation of the relaxed-ring is well defined in the following theorem.

THEOREM 3.2 *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 3 *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 3.2 clearly states the limitation of branches in the systems, helping developers to identify the scenarios requiring special failure recovery mechanisms. Since the problem is related to network partitioning, there seems to be no easy solution for it. An advantage of the relaxed-ring topology is that the issue is well defined and easy to detect, improving the guarantees provided by the system in order to build fault-tolerant applications on top of it.

4. Conclusion

The amount of Grid systems built on top of peer-to-peer networks is increasing. Since Grid users design their application at a higher level, it is reasonable to assume that failure handling will the delegated to the peer-to-peer system. This is why its crucial to provide a robust fault-tolerant network.

In this paper we have presented a novel relaxed-ring topology for fault-tolerant and self-organising peer-to-peer networks. The topology is derived from the simplification of the join algorithm requiring the synchronisation of only two peers at each stage. As a result, the algorithm introduces branches to the ring. These branches can only be observed in presence of connectivity problems between peers, allowing the system to work in realistic scenarios, providing fault-tolerant ring maintenance.

The guarantees and limitations of the system are clearly identified and formally stated providing helpful indications in order to build fault-tolerant applications on top of this structured overlay network. Having these guarantees, solving issues related to network partitioning become more addressable.

References

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

[2] Denis Caromel, Alexandre di Costanzo, and Christian Delbé . Peer-to-peer and fault- e tolerance: Towards deployment-based technical services. Future Generation Computer Systems, 2007. To appear.

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

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

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

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

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

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

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

[10] 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.

[11] Domenico Talia, Paolo Trunfio, Jingdi Zeng, and Mikael Heqvist. A peer-to-peer framework for resource discovery in large-scale grids. In Proc. of the 2nd CoreGRID Integration Workshop, pages 249-260, Krakow, Poland, October 2006.

[12] Paolo Trunfio, Domenico Talia, Paraskevi Fragopoulou, Charis Papadakis, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Seif Haridi. Peer-to-peer models for resource discovery on grids. In Proc. of the 2nd CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, Paris, France, January 2006.