

Improving Robustness in P2PS and a Generic Belief Propagation Service for P2PKit

Boris Mejías, Yves Jaradin, and Peter Van Roy

Université catholique de Louvain, Louvain-la-Neuve, Belgium
firstname.lastname@uclouvain.be

1 Introduction

Structured peer-to-peer networks are the most successful implementation of decentralised structured overlay networks. We develop P2PS [1], a Chord [2] alike peer-to-peer network with improvements in the ring maintenance and the routing algorithm. The construction of routing tables and the routing of the messages follow the Tango protocol [3]. P2PS implements a self-organising structured overlay network forming a ring of peers without central point of failure. To exploit such a network we use P2PKit [4], a library that provides a publish/subscription service mechanism.

This document presents the work that has been done in order to improve the robustness of the P2PS system, and the work done in the design and implementation of a generic Belief Propagation service using P2PKit. This work has been presented as a Demo in the CoreGRID Industrial Conference [5], in Sophia-Antipolis, France, November 2006. The following sections briefly introduce P2PS and P2PKit in order to have the basic knowledge to understand the work presented in this technical report.

1.1 The P2PS system

The P2PS system builds a network in the form of a ring where every peer knows its successor and predecessor. Every peer has a key as identifier, and it is also responsible for all the keys in the range started by the key of its predecessor, and ended by its own key. The range does not include predecessor's key. When a lookup is performed searching for key k , the peer responsible for this key must handle the request. Every time a node joins or leaves, the successor and predecessor pointers are updated to guarantee consistency of lookup requests. To make lookup more efficient, every peer is provided with a finger table. Every finger points to other peers in the network following the Tango protocol [3], allowing messages to be routed in a very efficient way, instead of just following successor and predecessor pointers. A P2PS network looks as it is depicted in figure 1.

This document will be focused in the maintenance of predecessor and successor pointers, without analysing the efficiency of the routing mechanism. We will see what happens when several peers try to join or leave the network at the same time.

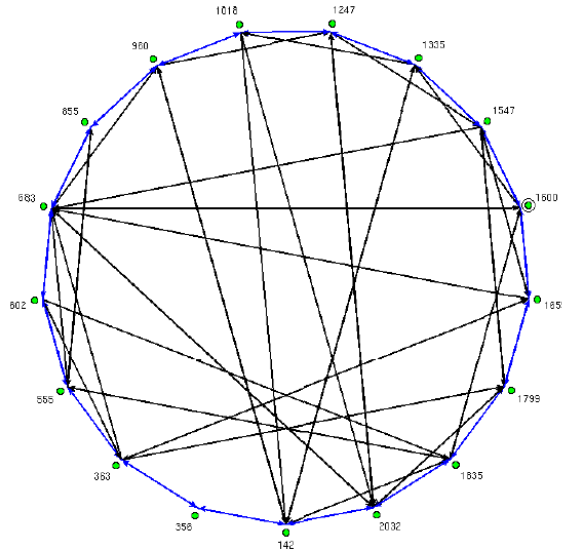


Fig. 1. A P2PS network.

1.2 The P2PKit library

P2PKit is a library that exploits the P2PS network with a service oriented architecture, making easier to implement decentralised applications on top this structured overlay network. A P2PKit client must connect to an existing peer in the network, which is responsible for routing the messages send by the client, and for delivering to the client the interesting messages that it receives. A client may subscribe to one or more services offered in the network. Messages belonging to these services are interesting for the client. A client may also install services to be read by other peers. Every service can be dynamically installed and-or upgraded. Figure 2 shows how a client connects to a peer in the network. P2PKit will be used to implement a generic Belief Propagation Service as it will be described in sections 4 and 5.

1.3 Structure of the document

The current join and leave algorithms for the ring maintenance of P2PS will be presented in the following section. Then, in section 3, we will briefly describe the atomic join and leave algorithm presented in [6], which will be implemented in our system. Sections 4 and 5 are dedicated to the generic belief propagation service designed using P2PKit, showing an example of usage with the k -partition algorithm for load balanced data retrieving. Conclusions and future work are presented in section 6.

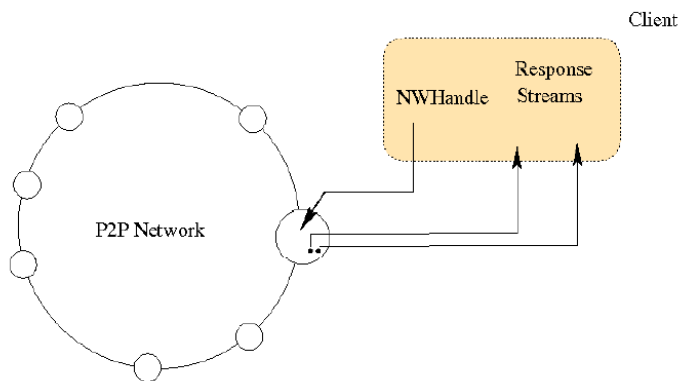


Fig. 2. A P2PKit client connected to a peer in the network.

2 Ring maintenance in P2PS

The current ring maintenance of P2PS has never been proved to be correct, and unfortunately, inconsistencies in successor and predecessor pointers has been observed during experiments done with the network. In this section we will present the algorithm for joining and leaving using event-driven notation. An example of the algorithm working will be followed by a counterexample proving that the algorithm is not correct for all cases, in particular when three or more nodes try to join the ring at the same time. This is the reason to replace the current algorithm by an atomic join and leave.

2.1 The event-driven notation

The event-driven notation will be used to describe the current join and leave algorithms of the P2PS system. They are used in the book “Introduction to Reliable Distributed Programming” [7] describing asynchronous distributed algorithms. Its use will be useful to compare P2PS algorithms with atomic join and leave described in [6], which are also written using this notation.

It is assumed that there is an event handler per each message sent in the peer-to-peer network. The actions taken by the peer are described inside the event, and they can involve local updates and new message sent to other peers. As an example, the following event is used to describe a message *event_name* sent by node *m* and received by node *n*, having arguments *arg1* ... *argn*. One of the action is a message sending to node *k*, having the name *another_event* and the arguments *a1* ... *an*.

```

event n.event_name(arg1 ... argn) from m
  <some actions>
  sendto k.another_event(a1 ... an)
  <some other actions>
end

```

2.2 The current join algorithm

To be able to join the ring, a node only needs to have access to one of the peers in the ring, and hold a key that is not used by any peer in the network. The process starts with a lookup request from the joining node to its contact in the ring. The contact peer will propagate this request in the ring, and it will eventually hit the responsible of the key of the joining node. The responsible of the key will reply directly to the joining node giving information over its predecessor. The joining node will immediately update its pointers *pred* and *succ*, used for predecessor and successor respectively. Having joining node identified by *j*, the responsible of the key with *s* and its predecessor with *p*, the algorithm carries on as follows.

- Node *j* updates its *pred* and *succ* pointers and notify *s* about its existence.
- When *s* receives the notification, verifies if *j* is a better predecessor than *p*, and replies to *j* with information over *p* and its successor list.
- Node *j* contacts node *p* informing that it is its new successor.
- Node *p* updates its *succ* pointer to *j* and notifies its predecessors, so they can update their successor lists.

No more messages are exchanged between peers *p*, *j* and *s*. Peer *j* has joined now the ring. The algorithm is presented now in detail using event-driven notation. Note that in almost every event there is a verification of the validity of the arguments as better successor or predecessor.

```
event n.reply_lookup_for_me(p, s) from m
  succ := s
  pred := p
  sendto succ.notify_by_pred()
end event

event n.notify_by_pred() from m
  if !betterPredecessor(m) then
    sendto m.bad_notify(pred)
  else
    oldpred := pred
    pred := m
    sendto m.succlist(con(n, sl), oldpred, n, slsize)
  end if
end event

event n.bad_notify(p) from m
  if betterSuccessor(p) then
    succ := p
    sendto succ.notify_by_pred()
  else % Insist with the same successor
    sendto succ.notify_by_pred()
  end
end event
```

```

event n.succlist(l, p, orig, i) from m
  sl := l
  if betterSuccessor(m) then
    succ := m
  end if
  if p != nil and betterPredecessor(p) then
    pred := p
    pred.succlist(con(n, sl), nil, n, slsize)
  else if i > 0 and orig != n then
    pred.succlist(con(n, sl), nil, orig, i-0)
  end if
end event

function n.betterSuccessor(m) returns boolean is
  returns m belongsTo (n, succ]
end function

function n.betterPredecessor(m) returns boolean is
  returns m belongsTo [pred, n)
end function

```

2.3 How it works

Let the network be formed by nodes $\{0, 3, 10, 16\}$. Then, three new nodes holding keys 4, 7 and 9 try to join at the same time. For the three of them, the node 10 is found to be the responsible of their keys after doing the lookup. Peers 0 and 16 are just included to avoid confusion with respect to predecessor and successor pointers in peers 3 and 10. There are two reasons for using 3 joining peers in this example. One is to show more variants of the problem, and the second reason is that the algorithm only fails when three or more peers try to join simultaneously, but that case will be described in the next section. For the variants of the problem, first it is shown what happens when the node with the smaller key joins first (node 4 before 9). Then, it is shown how the node with the bigger key joins first (node 9 before 7).

Let us see how the events are triggered. As a result for the lookup requested by nodes 4, 7 and 9, the following three events will be triggered:

4.reply_lookup_for_me(3, 10) from 10

7.reply_lookup_for_me(3, 10) from 10

9.reply_lookup_for_me(3, 10) from 10

After these three events, the three joining peers will have node 3 as predecessor and node 10 as successor, but they are still not part of the ring, because peers 3 and 10 still point each other as successor and predecessor. The situation is depicted in figure 3. Nodes having grey colour belong to the ring. Pointers between nodes 0 and 16 are not drawn.

Let us consider now that peer 4 is the first one in performing the join. The following events will be triggered:

10.notify_by_pred() from 4: Since 4 is a better predecessor than 3, peer 10 will updated *pred* to 4, and reply to the joining peer with the *succlist* event.

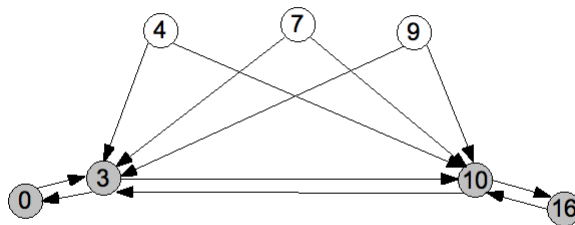


Fig. 3. Nodes 4, 7 and 9 try to join the ring simultaneously between nodes 3 and 10.

4.succlist(sl10, 3, 10, listsize) from 10: This event will confirm peer 4 that its *pred* and *succ* pointers are correct (3 and 10 respectively). Now, to finally enter into the ring, it needs to communicate peer 3 about its existence. The arguments *sl10* and *listsize* corresponds to the successor list of peer 10.

3.succlist(sl4, nil, 4, listsize) from 4: Since peer 4 is better successor than peer 10, *succ* pointer of peer 3 will be updated to 4. Now peer 4 has correctly joined the ring. The sequence $3 \leftrightarrow 4 \leftrightarrow 10$ is now part of the ring as it is depicted in figure 4.

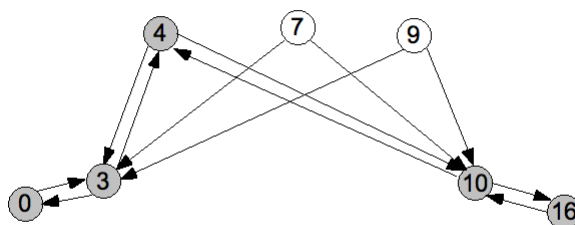


Fig. 4. Nodes 4 correctly joined the ring.

Now, let us see how peer 9 enters into the ring. Remember that after event *9.reply_lookup_for_me(3, 10)* is triggered, pointers *pred* and *succ* of node 9 are set to 3 and 10. Then, the following events are triggered.

10.notify_by_pred() from 9: Peer 9 is better predecessor than 4, then, *pred* at peer 10 is updated to 9. Peer 10 replies to peer 9 with event *succlist*.

9.succlist(sl10, 4, 10, listsize) from 10: Peer 4 is better predecessor than 3, then, *pred* at peer 9 is updated to 4. This event and the following one are crucial for the success of the algorithm. Note that before this event is triggered, the ring has the following sequences: $3 \leftrightarrow 4 \rightarrow 10$ and $3 \leftarrow 9 \leftrightarrow 10$. We will see later how the algorithm can fail at this point.

4.succlist(sl9, nil, 9, listsize) from 9: Peer 9 is better successor than 10, then, *succ* is updated to 9 at peer 4. Now we have a ring again with the sequence $3 \leftrightarrow 4 \leftrightarrow 9 \leftrightarrow 10$.

The joining process is a little bit extended for node 7. In the beginning, it received nodes 3 and 10 as predecessor and successor. Then, the following event is triggered.

10.notify_by_pred() from 7: According to peer 10, peer 7 is not better predecessor than 9. As a consequence, the *bad_notify* event is triggered as reply of node 10.

7.bad_notify(9) from 10: Peer 9 is better successor for 7, so it updates its *succ* pointer to 9, and sends once again the *notify_by_pred* event, but this time to peer 9.

9.notify_by_pred() from 7: Now peer 7 is better predecessor than peer 4, so pointers are updated at peer 9 and the algorithm continues as it was described for peers 4 and 9. Events *7.succlist(sl9, 4, 9, listsize) from 9*, and *4.succlist(sl7, nil, 7, listsize) from 7*, will finally join peer 7 into the ring. The ring has now the sequence $3 \leftrightarrow 4 \leftrightarrow 7 \leftrightarrow 9 \leftrightarrow 10$ as it is depicted in figure 5.

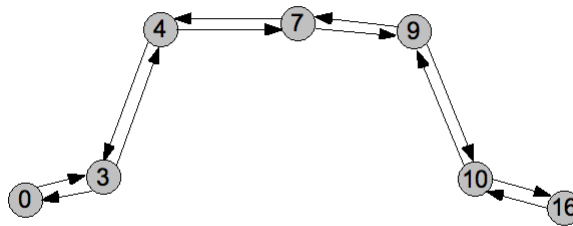


Fig. 5. Nodes 4, 7 and 9 have successfully joined the ring.

2.4 How it fails

Using the same example presented in the previous section, we will slightly modify the order of the events to show how the algorithm can fail. The initial scenario consist of a network formed by nodes $\{0, 3, 10, 16\}$, and three joining nodes holding keys 4, 7 and 9. The first events to be triggered are the following.

4.reply_lookup_for_me(3, 10) from 10

7.reply_lookup_for_me(3, 10) from 10

9.reply_lookup_for_me(3, 10) from 10

We will assume that node 4 is the first one joining the network. Events *10.notify_by_pred()* from 4, *4.succlist(sl10, 3, 10, listsize)* from 10 and *3.succlist(sl4, nil, 4, listsize)* from 4 are triggered consecutively and node 4 correctly joins the ring. Now the network contains nodes {0, 3, 4, 10, 16} as it is shown in figure 4. Nodes 7 and 9 are trying to join having peers 3 and 10 as predecessor and successor. Now node 9 starts joining the ring.

10.notify_by_pred() from 9: Node 9 is better predecessor than 4. This event will update pointer *pred* to 9 in peer 10. Now we will change a bit the order of the events with respect to the previous section. Instead of carrying on with the events associated to peer 9, we will see how peer 7 starts its joining process.

10.notify_by_pred() from 7: Node 7 is not a good predecessor for 10, because *pred* is pointing to 9 at this moment. The event *bad_notify(9)* is triggered at node 7.

7.bad_notify(9) from 10: Peer 7 updates *succ* to 9, and it informs peer 9 about its existence.

9.notify_by_pred() from 7: Peer 7 is better predecessor than 3, then, peer 9 updates its pointer *pred* to 7. Note that this time node 9 know nothing about node 4, and neither does node 7.

7.succlist(sl9, 3, 9, listsize) from 9: Peer 7 checks that 3 and 9 are correct predecessor and successor, but this check is wrong. At this moment the ring has the following sequences: $3 \leftrightarrow 4 \rightarrow 10$ and $3 \leftarrow 7 \leftrightarrow 9 \leftrightarrow 10$. This is depicted in figure 6. The ring is broken at the moment.

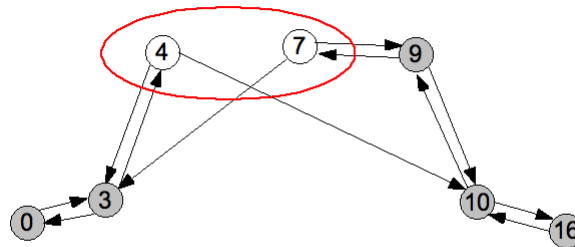


Fig. 6. Nodes 4 has a wrong successor and node 7 has a wrong predecessor.

3.succlist(sl7, nil, 7, listsize) from 7: Peer 7 notifies peer 3 about its existence, but peer 3 does not fix its *succ* pointer because 4 is better successor than 7. The ring remains broken, and peer 7 consider itself as being part of the ring correctly.

Let us come back now to the joining events of peer 9, who now receives the answer from peer 10 to the *notify_by_pred()* event.

9.succlist(sl10, 4, 10, listsize) from 10: Peer 9 considers that peer 4 is not a better predecessor than 7, and then, it does not update its pointer. Peer 9 has correctly joined the ring, and this event does not help peer 7 to fix its pointers. The ring remains broken. No other events updating *pred* and *succ* pointers will be triggered. The joining process has failed.

This counterexample proves that the join algorithm is not correct for all cases. The inconsistency it is generated by two main issues: when node 7 notifies its successor, peer 9, this one does not know about the existence of peer 4, and then, peer 9 still thinks that peer 3 is the right predecessor for peer 7. Then, when peer 7 informs peer 3 about its existence, peer 3 discards it because it already has peer 4 as successor. The second key issue is that when peer 9 receives notification about the existence of peer 4, it just discards it because it is using peer 7 as predecessor.

The problem with the algorithm is that it is not atomic. The events of two joining peers can interleave updating *pred* and *succ* pointers. These partial updates might also generate inconsistencies with lookup requests performed while the peers are joining. Another consideration is that the algorithm does not use any confirmation from the predecessor to its original successor. This confirmation from the predecessor can be used as ending statement of the joining operation. Having that confirmation, any update of *pred* and *succ* pointers must be delayed until the joining operation has finished.

2.5 The leave algorithm

We consider that the current leave algorithm it is implemented in a simplistic way. The leaving peer just set its *pred* and *succ* pointers to nil. Then, the peer leave the network without notifying any other peer. The ring is repaired by correction on use. The algorithm is described with the following event.

```
event n.leave_net() from n
    succ := nil
    pred := nil
end event
```

Since there is no notification to the successor or predecessor, there is no guarantee that the current ring is actually correct. The only advantage of this approach is that the peer can leave as fast as possible, and if the algorithm works well for a regular leaving peer, it will also work in case of failure of the peer. No deeper analysis will be done for the leaving algorithm, because the new implementation of atomic join will also imply modification in the leaving algorithm.

3 The atomic join and leave algorithm

The algorithm that will be implemented for the P2PS library is taken from Ali Ghodsi's PhD. dissertation, "Distributed *k*-ary System: Algorithms for Distributed Hash Tables" [6], chapter 3. It is not the intention of this report to describe the details of this algorithm.

Only the general idea will be presented in order to give the reader a notion of how the algorithm works. Further details can be found in the dissertation.

The algorithm is based in a simple locking system. To join or leave, every peer needs to get its own lock and the lock of its successor. Given this locking mechanism, the algorithm makes a difference between graceful and non-graceful leaves. Non-graceful leaves can occur due to a crash of the peer or a network failure. This locking system is meant to guarantee atomicity, but it has the drawback that a leaving peer can not leave as soon as it should. The description of join and leave algorithm does not consider the failure of any of the participants. Non-graceful leaves are left for future work.

As we already mentioned, a peer needs to get its own lock and the lock of its successor in order to join or leave. In the case of a joining peer, there will be no problem getting its own lock because no one else knows about it. Once the joining peer gets the lock of its successor, its join is guaranteed to be performed before any of its successor or predecessor leaves, and before other nodes try to join in the same range of keys. In the case of a leaving peer, getting its own lock can be delayed due to its predecessor trying to leave, or due to a new peer trying to join having it as successor. Getting successor's lock can be delayed because the successor is trying to leave, or because a new peer is joining between them. In both cases, the peer might change its successor. Considering that, the lock request will be forwarded to the correspondent peer.

The algorithm guarantees safety and liveness properties. It is guaranteed to be free of deadlocks, livelocks and starvation. Deadlock can occur when all the peers want to leave at the same time. This situation is similar to the "*Dining Philosophers Dilemma*" because every peer will take its lock, but it will not be able to take its successor lock. To avoid deadlocks, the peer with the greatest key will try to get its successor's lock first, and then its own lock. To ensure starvation freedom, which is a stronger property than no livelocks, the algorithm uses a forwarding mechanism for locks request. Every time that a node starts its leaving process, it forwards any lock request to its successor. Like that, every peer is ensured to make progress acquiring locks.

The description of the algorithm using event-driven notation can be found in pages 66, 67, 70 and 71 of Ghodsi's dissertation [6]. The *joining* part works basically as follows, having node j as the joining peer, and nodes p and s as predecessor and successor.

- Node j acquires its own lock
- Node j sends a join request to node s
- If the lock of s is not taken yet, it is granted to j , and s updates its *pred* pointer.
- Node s informs node j about predecessor p .
- Node j updates its *succ* and *pred* pointers, and informs peer p about its existence.
- Node p updates its *succ* pointer and informs s about this change.
- When node s receives this acknowledgement from peer p , it notifies node j that the joining process is done.
- When node j receives this notification, then the joining has really finished.

A considerable difference between this algorithm and the current one, is that pointers are only updated when locks are acquired, with the only exception of *succ* pointer of node p . In the case of that update, there is also an acknowledge message sent from p to s that guarantees the validity of this update.

For the *leaving* part of the algorithm, let us consider the node l as the leaving peer, and nodes p and s as its predecessor and successor. The algorithm works as follows:

- Peer l acquires its own lock and sends a lock request to s
- Once s 's lock is granted to l , l sends a leave request to s giving p as new predecessor.
- Node s notifies p that it will become its new successor.
- Node p updates its pointer, and as in the *join* algorithm, when a node is notified about a new successor, it sends an acknowledgement to its current successor, in this case node l .
- When l receives this acknowledgement, it knows that s and p has correctly updated their pointers, and it is ready leave sending a last notification to s confirming that the leaving process is done.

No correction of relevant pointers is needed after the node l leaves, and the lookup requests are guaranteed to be correct at any moment. That is the main advantage with respect to the current leave algorithm of P2PS. One of the drawback of this algorithm is that a peer is not allowed to leave until it acquires the relevant locks. This could be a problem if a user is trying to shut down its system and it is delayed due to a graceful leaving of her peer-to-peer application. Another main drawback is that more work needs to be done in order to handle non-graceful leaving, which at the moment are let for future work.

4 Belief Propagation Service

The belief propagation service is inspired by the classical belief propagation algorithm using the overlay topology as the graphical model. The service support multiple clients, conducting unrelated belief propagations in parallel. Each of these has an identifier. The description below is for a single client.

4.1 Description of the algorithm

The belief propagation algorithm [8] is used to compute marginal probabilities or most likely assignment for certain factorised distributions. If we have n random variables X_1, \dots, X_n , and a distribution of the form

$$P(X_1 = x_1, \dots, X_n = x_n) = P_1(x_1) \cdots P_n(x_n) \cdot \prod_{1 \leq i, j \leq n} P_{ij}(x_i, x_j)$$

we create a graph with n nodes (corresponding to the n random variables and associated P_i) and an edge between node i and node j if the function P_{ij} is not constant (corresponding to the non-trivial P_{ij})

The algorithm is step based. At step t , node i sends to connected node j the message $m_{ij}^{(t)}$ which describes a distribution for X_j computed as:

$$m_{ij}^{(t+1)}(x_j) = \alpha \max_{x_i} P_{ij}(x_i, x_j) P_i(x_i) \cdot \prod_{k \in N(i) \setminus \{j\}} m_{ki}^{(t)}(x_i)$$

where $N(i)$ denotes the indexes of nodes connected to node i and α is a normalisation factor.

It should be noted that a node with only one neighbour will send the same message at each step and by induction, in a tree, the messages from i to j are identical at each step after the one where the message appears for the first time.

Each node can compute its belief using the formula:

$$bel_i^{(t)}(x_i) = \alpha P_i(x_i) \prod_{j \in N(i)} m_{ji}^{(t)}(x_i)$$

This algorithm is interesting for distributed peer-to-peer applications because if the graph happens to have the structure of the peer-to-peer network, we can use plain message passing to compute a global maximum for this statistical distribution.

4.2 Adaptation to P2PS

The main differences between the P2PS routing graph and the graphs described above is that in P2PS we have a directed graph. Because of this, we will only send messages along the edges going in the right direction. We redefine the $N(i)$ function to return the nodes which have a finger to node i , and we remove the set subtraction of node j in the message formula as it would remove j only if there are arrows in both directions between node i and node j , and in this case, it makes sense to consider the “two” j as separate nodes.

We now have that

$$m_{ij}^{(t+1)} = \alpha \max_{x_i} P_{ij}(x_i, x_j) bel_i^{(t)}(x_i)$$

and also that

$$bel_i^{(t+1)}(x_i) = \alpha P_i(x_i) \prod_{j \in N(i)} \max_{x_j} P_{ji}(x_j, x_i) bel_j^{(t)}(x_j)$$

We can then simply use the current belief as a message.

Since the graph is not a tree, we need to start the ball by using an initial belief. This is created on each node using user-supplied code

According to the formula, we need to wait for messages from all the incoming edges. This is difficult to detect exactly since a node does not know which nodes point to it. Up to now, we used a simple timeout (reset at each incoming message) to decide the probable completion of the step, but this can easily be improved to a more adaptive solution.

Respecting peer-to-peer principles, we consider all the P_i and all the P_{ij} (corresponding to fingers) to be equal. As P2PS uses a Chord-like structure, we could have used different functions for the particular predecessor and successor pointers, but this has not been done up to now.

To improve convergence, we also used a variant where the $P_i(x_i)$ in the belief update formula is replaced by $bel_i^{(t)}(x_i)$. If the initial belief is chosen according to P_i this corresponds to decrease the influence of this factor over time to reduce the instability in the beliefs.

At any point in time, any node can be queried for its current belief.

4.3 Usage of the service

The generic belief propagation service is a regular P2PKit service. It is published using the *installService* primitive of P2PKit. It can be marked as copyable, so the installation has to be done just once per network, regardless of churn, provided the service is used. The service recognises a *nop* message which has no effect, and that can be used by a keep-alive service to ensure some activity for the service.

A P2PKit client can trigger the belief propagation algorithm with the *init* message which takes as arguments:

1. An identifier for this belief-propagation, typically an Oz name.
2. The edge potential. All edges have the same potential. A further enhancement could give different potentials to the successor and predecessor edges.
3. the self-potential, where *unit* can be given to use the previous belief as the self-potential
4. the initial-belief maker. This is a function which takes an environment, identical to the one passed by P2PKit to a service maker. It allows the initial-belief maker to use resources local to the node, such as a pseudo-random numbers generator.
5. the maximum number of messaging rounds. This is because the belief-propagation algorithm can have convergence problems on a graph.

Since we are using belief propagation on a graph, there is no guarantee of convergence. The maximum number of rounds is thus an important parameter as it forces the algorithm to finish in a finite time.

The client can query the service at any time using the *getCurBelief* RPC message, which takes only the identifier for the belief propagation the client is interested in, and it returns the current belief of the node and the step number it reached.

A different message *getCurBeliefBCast* takes as second argument a P2PKit access point to which the answer will be sent. This message can be used when RPC is impractical to use, e.g., when we are interested in all the nodes' beliefs, it's easier to broadcast a message rather than query each node. RPC broadcast is not implemented in P2PKit.

5 Application to k -partitioning

The k -partitioning problem is a typical example of why belief-propagation based algorithm might be useful in an overlay network. The problem is to assign each node of the network to one of k partitions while having a certain (fixed) proportion of nodes in each partition and avoiding adjacent nodes being in the same partition. This algorithm is useful for DHT with an imperfect hash, which is needed if we want to express advanced queries such as range queries. As an example, to store an English dictionary, we could use the first letter as the partition where the word has to be stored. Quite evidently, some partitions need to be bigger than others. To reduce hot spots in the network, we also want to spread the partitions over the network and so we exclude adjacent nodes from being in the same partition. Example taken from [9].

This problem translates to belief-propagation in a straightforward way. The self-potentials are the distribution of partitions that we want. The edge potential is a matrix

with zeroes on the diagonal and constant values everywhere else. The initial belief is drawn randomly for each node using the self-potential as the distribution.

So if we want k partitions with a probability p_a to be in partition a , we will use $P_i(a) = p_a$, $P_{ij}(a, a) = 0$ and $P_{ij}(a, b) = 1$ if $a \neq b$. The initial belief at node i is $bel_i^{(0)}(a_i) = 1 - (k - 1)\epsilon$ for a certain a_i and a small fixed ϵ , and $bel_i^{(0)}(b) = \epsilon$ for $b \neq a_i$. The a_i is chosen independently for each node according to the distribution P_i . This gives better results than just using P_i as it creates more asymmetry in a system which is otherwise extremely symmetric (because nodes are peers). The algorithm is based on [9].

It should be noted that to be able to converge, the number of partition must be (much) greater than the degree of the nodes. Like that, it is relatively easy for adjacent nodes to be in different partitions. Also note that the number of nodes must be (much) greater than the number of partitions, and then, each partition is sufficiently populated to approach the expected distribution. This is not a problem as in theory the degrees of nodes in a P2PS network is in the order of the logarithm of nodes. But, for small networks, which are the only ones that can be represented graphically in a demonstration, these numbers tend to be too close. A solution is to use the previous belief as the self-potential so that the proportion of each partition requirement is less strongly enforced.

5.1 Graphical interface

In order to demonstrate our service, we wrote a small application for k -partitioning that interactively shows the beliefs of all the nodes in the network as pie charts (see figure 7). This allows us to see the algorithm converging in real time. If the user wants to see a smooth evolution of the algorithm, it is necessary to slow down the application. The test case also allows us to observe the influence of the network size and parameters, as well as the influence of the unevenness of the distribution of the partitions and the number of partitions.

6 Conclusion and Future Work

The work presented in this technical report is divided in two parts: the analysis of the ring maintenance of the P2PS network, and the design and implementation of a generic belief propagation service in P2PKit. The ring maintenance has been studied through the join and leave algorithms. We have shown that the join algorithm can fail when three or more nodes try to join the ring at the same time, proving that the algorithm is not correct for all the cases. The leave algorithm has not been deeply studied, but it is considered to be a bit too simplistic. As a solution for the ring maintenance, an atomic join and leave algorithm is implemented in P2PS, following the design described in section 3. More work needs to be done in the case of failures of peers.

The generic belief propagation service that has been implemented using the service architecture of P2PKit, has allowed us to implement the k -partitioning algorithm described in section 5. This algorithm divides the network into k different sets respecting

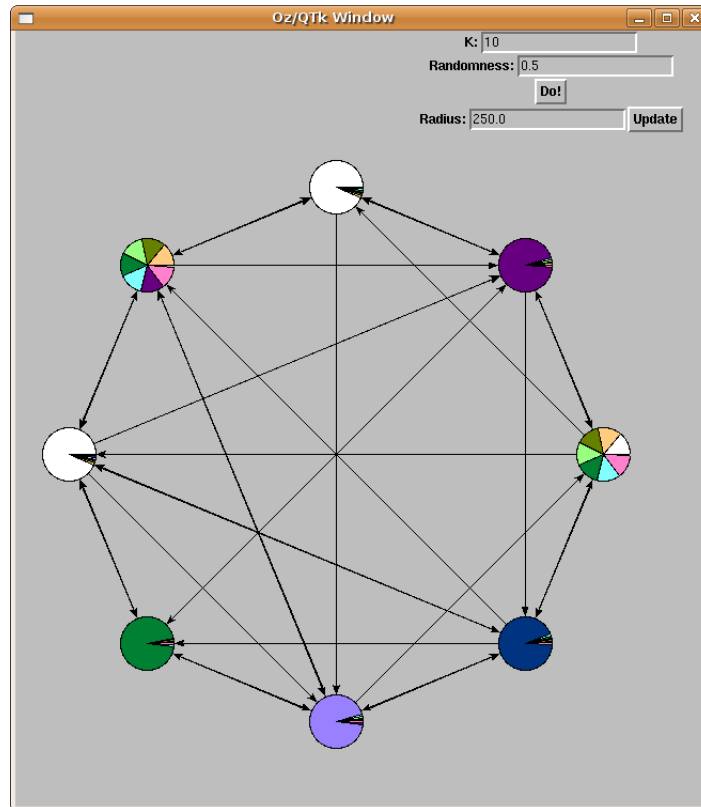


Fig. 7. The test application running.

load balancing constraints for data retrieving. This work is a first step in order to use belief propagation based algorithms to improve self-organisation of the structured overlay network. The next step in this direction is the churn estimation using Gaussian belief propagation. A critical churn value can be used to trigger correction of routing tables.

7 Acknowledgements

We would like to thank Kevin Glynn for the insights he has provided to understand the design and implementation of P2PS and P2PKit. We also want to thank Danny Bickson for his explanations about Belief Propagation algorithms. This research is partly supported by the projects CoreGRID (contract number: 004265) and EVERGROW (contract number:001935), funded by the European Commission in the 6th Framework programme.

References

1. Mesaros, V., Carton, B., Glynn, K.: P2PS: A peer-to-peer networking library for mozart/oz. <http://p2ps.info.ucl.ac.be/index.html> (2006)
2. Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: ACM SIGCOMM 2001, San Diego, CA (2001)
3. Mesaros, V., Carton, B., Van Roy, P.: P2PS: Peer-to-peer development platform for mozart. In Van Roy, P., ed.: MOZ. Volume 3389 of Lecture Notes in Computer Science., Springer (2004) 125–136
4. Glynn, K.: P2PKit: A services based architecture for deploying robust peer-to-peer applications. <http://p2pkit.info.ucl.ac.be/index.html> (2006)
5. Jaradin, Y., Mejías, B., Van Roy, P.: Applying decentralized algorithms in peer-to-peer networks (2006)
6. Ghodsi, A.: Distributed k -ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden (2006)
7. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer-Verlag, Berlin, Germany (2006)
8. Yedidia, J.S., Freeman, W.T., Weiss, Y.: Understanding belief propagation and its generalizations. (2003) 239–269
9. Bickson, D., Dolev, D., Weiss, Y., Aberer, K., Hauswirth, M.: Indexing data-oriented overlay networks using belief propagation. In: In the 7th Workshop of distributed algorithms and data structures (WDAS 06). (2006)