Core-based Reconfiguration for Reliable Overlay Networks

Silvia Bonomi
Sapienza Univertità di Roma
Dipartimento di Informatica e Sistemistica
Via Ariosto 25, 00185 Roma, Italy
bonomi@dis.uniroma1.it

Sara Tucci-Piergiovanni
Sapienza Univertità di Roma
Dipartimento di Informatica e Sistemistica
Via Ariosto 25, 00185 Roma, Italy
tucci@dis.uniroma1.it

ABSTRACT

In P2P overlay networks, the continual arrival and departure of nodes leads to the necessity of running periodically a specific reconfiguration algorithm able to renew the overlay. This paper presents a new reconfiguration mechanism for overlay networks. The algorithm exploits a characteristic observed in many overlay networks: the presence of a core composed by reliable nodes (nodes that never leave the overlay). Even if members of the core are a-priori unknown, the algorithm is able to eventually select one of them that will carry out the periodic renewing of the overlay. Once the reconfiguration is performed by a core node, the overlay will never lose its reliability, letting any query, issued from this time on, be ever satisfied.

1. INTRODUCTION

Recently, Peer-to-Peer (P2P) systems attract attention. In P2P systems, each user shares and exchanges information as equals by playing the role of both the server and the client. Because nodes that are participating to the system are connected to each others to construct an overlay network, the p2p paradigm is superior on scalability compared to client-sever model. However, P2P networks have technical issues that should be solved. Generally, P2P networks are categorized into two types, structured [6, 7, 8] and unstructured [1, 2, 3, 9]). Due to the absence of a server node that centrally stores and manages all of the information, a query has to be routed inside the network to search the a-priori unknown set of nodes that maintain the matching information. Therefore, a lot of effort has been paid in order to realize efficient and reliable information search/retrieval mechanisms by properly constructing the overlay network with the aim of improving successiveness of the search, shorten of search delay, or reducing a total resource consumption. The issue of reliability is particularly challenging in P2P systems due to the continuous arrival and departure of nodes. This phenomenon, also called churn, may seriously compromise overlay functionalities. For instance, overlay may degrade until disconnection as time passes by, avoiding queries to correctly search for the matching results.

To face this problem many approaches have been proposed. In structured overlay networks, a dedicated routine is in charge of revealing node departures [8]. Upon the detection of a node departure, this routine starts an overlay reconfiguration devoted to renew the overlay network, in order to come back to the operative state. In unstructured overlay networks, usually there is no dedicated routine to detect departures [9]. The departure detection happens only when a node wants to exchange information with an already departed one. Usually, the detection does not lead to renew the entire overlay, but who detected the departure only removes its pending connection. Note that the approach used in unstructured overlays can be pursued because a departure affects only locally the overlay, in contrast with structured overlays where a departure may affect the whole overlay. Let us note that if a departure affects the whole overlay, a possible burst of departures may significantly lengthens the time for the reconfiguration to take effect. The longer the time is for reconfiguration, the higher is the number of queries that can cross a still damaged network with the possibility to return an empty (or partial) result. This could lead to a very poor reliability.

The approach proposed in this paper circumvents the drawbacks of a continual global reconfiguration starting from the observation that not all nodes in the network have the same lifetime. It is widely recognized that overlay networks contains, among a huge number of nodes with very short lifetimes, a *core*, i.e. a small number of very reliable nodes with unlimited lifetime [5]. However, the identities of the core members are not a priori known.

In this paper we propose a *core-based* approach to overlay reconfiguration. The basic idea is to eventually elect a core node as a special node able to carry out the reconfiguration. This node, also called *supervisor*, actually carries the reconfiguration in a proactive way: the node renews the overlay periodically, before it degrades to disconnection. This can be done deterministically, by knowing an upper bound on the departure-rate of nodes. More in details, the supervisor periodically gathers information about current alive nodes inside the network, computes for each node a new state (e.g. new connections to other nodes) and communicates it to these alive nodes, which update their state before the network is damaged; in such a way a query crossing the network at any time will be ever satisfied. Note that, at the beginning, the

algorithm is not able to give to a core node the supervisor role, as core nodes are unknown to the algorithm. However, the proposed algorithm is able to converge to the election of a core node as supervisor after a finite time. Before this time, queries can be lost, but after this time, queries will be satisfied forever, contrarily to what happens in the structured approach.

The paper is organized as follows: Section 2 defines the system model and Section 3 presents the reconfiguration algorithm along with its correctness proof.

2. SYSTEM MODEL

We consider an infinite set of processes $\Pi=\{p_1, p_2 \dots, p_n \dots\}$. Each process has a unique identifier for all its life time in the system. Processes are arranged in a logical network, called *Overlay Network* (ON), built on top of the physical one. The ON can be seen as a graph where each node represents a process $p_i \in \Pi$ and each edge represents a logical communication channel between two elements $p_i, p_j \in \Pi$ such that p_i and p_j can communicate. Every process is able to communicate with its direct neighbors in the ON by means of messages exchange on point-to-point reliable channels. There exists a known bound δ on the message transmission delay on the considered channels, hence the system can be considered as *synchronous*. We define as *correct* a process that never fails. A faulty process fails by crashing and if it recovers from the crash then it is considered as new in the system (with a new identifier). System is *dynamic*, i.e., nodes may join and leave system at any time. More formally, we assume that:

- when a node leaves the system, it does not perform any specific task and then a voluntary leave is considered as a failure. In the following we consider equally faults and voluntary leaves and we refer them as faults;
- when a node fails, it is removed from the graph together with all its incident edges;
- a node joins the system through a join event. The overlay is actually composed by all and only those nodes that have joined the system and have not left yet.
- there exists a known upper bound f_r on the node failure rate (i.e. the number of nodes that fail/leave the system in each time unit);
- there exists a-priori unknown finite set of processes, called *core* $\in \Pi$, that never crash or leave the system [4, 5].

3. THE RECONFIGURATION ALGORITHM

3.1 Algorithm Overview

The proposed algorithm aims at keeping overlay connectivity most of the time. Formally the problem is defined as follows:

PROPERTY 1. ***Eventual Connectivity.*** *Eventually and permanently, all processes in the ON will be included in a connected graph G_P.*

In our algorithm, a node, called *supervisor*, is elected and it will be in charge of running the reconfiguration procedure.

In a dynamic system, continuous failures of nodes make connectivity of the ON decreasing. In our approach, the ON graph is a k-connected graph so that it will be resilient to at least $(k-1)$ failures from its definition.

Fixing a desiderata degree of connectivity k and given the failure rate f_r, it is possible to know how long the ON graph will be connected.

PROPERTY 2. *Let k be the degree of connectivity and let t be the time where the ON graph is defined. Let be f_r the failure rate of the system, then the ON graph will be connected for a time period $\Delta T = (k-1)/f_r$ from t.*

We call this period *graph lifetime* L_G.

The idea of the protocol is to use the degree of connectivity and the failure rate to know when the overlay is becoming disconnected and renew it. This is the task of the supervisor that recomputes a new k-connected graph G_{new} and communicates it to all the other nodes in the system before the graph lifetime L_G of the current graph G expires. The idea of the reconfiguration protocol is the following: during the lifetime of G, the supervisor starts the reconfiguration procedure to compute G_{new}. The reconfiguration is composed of three phases: (i) nodes health verification, (ii) graph computation, (iii) graph dissemination. The first phase consists of a verification of the alive nodes in the system. During this phase, the supervisor sends a "ping" message to all the nodes included in the current graph G and to newly joined nodes, thus waits for their replies. Then the second phase starts and G_{new} is computed including all nodes which have replied. In the third phase, the supervisor starts the dissemination of the G_{new} to all the alive nodes included in the new graph.

If the supervisor does not crash, connectivity is preserved as the dissemination of the new graph G_{new} deterministically terminates before L_G expires.

If the supervisor crashes, a new election will start and a new supervisor is selected. During the period when the supervisor is changing, connectivity cannot be assured. However, due to the core assumption, we have that in a finite number of elections, and then in a finite time, a core node will become a supervisor and then connectivity will hold forever.

The join to the overlay is also managed by a specific procedure that allows a new node to become part of the ON by means of an access point node selected among the ones already connected in the ON through a graph computed by some supervisor.

3.2 Algorithm pseudo-code

3.2.1 Data Structures

At the process start-up, all the data structures have to be initialized. In Figure 1 the pseudo-code of the initialization phase is shown.

Init
1 $supervisor = nil$
2 $active = false$
3 $myLevel = 0$
4 $joined = \emptyset$
5 $myKnowledge[] = emptyArray$
6 $parentsLevel = 0$
7 $parents = \emptyset$
8 $candidates = \emptyset$

Figure 1: Data Structure Initialization

Data structures, maintained by a node p_i, collect several information, as described as follows:

The variable *active* is initially false and does not change until p_i does not install[1] the first graph and remains true until p_i crashes or leaves the system.

The variable *myLevel* represents a logical "distance" between p_i and the supervisor and is set during the join procedure depending from the access point used by p_i to join the ON.

The variable *myKnowledge* contains the information about processes currently joined to the ON. This information is actually gathered by the supervisor and it is the only one allowed to communicate using connections to processes stored in the knowledge variable. This communication happens when the supervisor has to communicate the new graph to current participants. This variable also maintains the information about the distance that processes have from the current supervisor. To this end the variable *myKnowledge* has an array structure defined as follows: at the i-th entry of the array it is stored the set of nodes having level i.

The variable *supervisor* contains the identifier of the node that is currently considered as supervisor for the ON by the process p_i.

The variable *joining* contains a set of nodes joined from the installation of the last graph and not yet active, including those that joined using p_i as access point.

The information used to elect a supervisor is stored in the variables *parents* and *candidates*; *parents* is the set containing all the nodes alive at the level just before p_i's level and represents the nodes to be monitored in order to detect a possible crash of the supervisor while *candidates* is the set containing the nodes at p_i level that are possible candidates for the election of the new supervisor. Due to the failures, an entry of the *myKnowledge* array may become empty and then there exist some nodes inside the system that have to update the level of their parents in the *myKnowledge* structure. To this aim we store in the variable *parentsLevel* the current level not empty where the node can find its parents in the *myKnowledge* structure.

3.2.2 Join Procedure

[1]A graph is installed when a node receives it and starts to communicate using its links.

We assume to have a bootstrap service that makes possible, for incoming nodes, to find an access point to the ON. In Figure 2 is presented the Join protocol.

Join(p_i)
1 **if** ($p_i == nil$)
2 **then**
3 $knowledge[0] = myId$
4 $supervisor = myId$
5 $active = true$
6 *set timerGraph* $= ((k-1)/f_r) - 4\delta$
7 **else** **send** ("$Join''$, $myId$) **to** p_i

(a)

when (receive ("$Join''$, i) **from** p_i) **do**
1 **if** ($supervisor \neq myId$)
2 **then** $joined = joined \cup \{p_i\}$
3 $knowledge[myLevel+1] = knowledge[myLevel+1] \cup \{p_i\}$
4 **send** ("Ack'', $myLevel$, $supervisor$, $knowledge[]$) **to** p_i

(b)

when (receive ("Ack'', $level$, s, $knowledge[]$) **from** p_i) **do**
1 $myLevel = level + 1$; $parentsLevel = level$
2 $myKnowledge[] = knowledge[]$
3 $supervisor = s$

(c)

Figure 2: Join Protocol

The joining node p_j contacts the bootstrap service that returns an active node p_i already member of ON if there exists, otherwise it returns *nil*.

If no process is returned by the bootstrap service, p_j starts to build the knowledge inserting itself at the level 0 and becoming active. At this point p_j is the first node part of ON. Since p_j is the only node part of the ON it becomes automatically the supervisor and then it starts to monitor the ON. In particular it sets a timer, namely *timerGraph*, equal to the graph lifetime L_G (minus the time needed to verify the health of the ON participants and to communicate the new graph before the current one lose the connectivity) and then starts the *Reconfiguration* procedure.

If a process p_i is returned, p_j contacts p_i sending a message of join request with attached its identifier. When p_i receives the request of p_j, it updates its knowledge (storing p_j identifier at its level plus one) and sends back to p_j an acknowledgment message containing p_i's level, the current supervisor and its knowledge. Moreover, if p_i is not the current supervisor, it puts the identifier of p_j in the list of joined nodes in order to let the supervisor aware of p_j for the next graph. When p_j receives the ack of p_i, it updates its structure with the information received.

3.2.3 Reconfiguration Procedure

The Reconfiguration procedure is managed by the supervisor node and it is triggered periodically.

In Figure 3 it is shown the reconfiguration protocol.

This protocol is based on the usage of two timers, *timerGraph* and *timerMonitor*, exploiting the synchrony of the system. *timerGraph* measures the graph lifetime while *timerMonitor*

```
1 when (timerGraph expired) do
2  for each (p ∈ memberOf(myKnowledge)) do
3   send ("ping") to pfor each i do
4   myKnowledge[i] = ∅
5  Set timerMonitor = 2δ
```

(a)

```
1 when (receive ("ping") from p_i) do
2  send ("pong", joined, myLevel) to p_i
```

(b)

```
1 when (receive ("pong", joined, level) from p_i) do
2  myKnowledge[level + 1] = myKnowledge[level + 1] ∪ joined
3  myKnowledge[level] = myKnowledge[level] ∪ p_i
```

(c)

```
1 when (timerMonitor expired) do
2  if (|member(myKnowledge[])| > f(k))
3     then G ← compute_graph(to_monitor ∪ myId, k)
4          for each p ∈ member(myKnowledge[]) do
5            send ("newGraph", G, myKnowledge[]) to p
6  Set timerGraph = ((k − 1)/f_r) − 4δ
```

(d)

```
1  when (receive ("newGraph", G, knowledge[]) from p_i) do
2   if (¬active)
3      then active = true
4           parentsLevel = max_i{i < myLevel∧
5           myKnowledge[i] ≠ ∅}
6           parents = parents ∪ knowledge[parentsLevel]
7           candidates = candidates ∪ knowledge[myLevel]
8           trigger monitorSupervisor()
9   myKnowledge[] = knowledge[]; supervisor = p_i; joined = ∅
10  install(G)
```

(e)

Figure 3: Graph Reconstruction Procedure

measures the time needed to collect all the information needed for the detection. When the *timerGraph* expires, the supervisor sends a ping message to all the nodes in *myKnowledge*, resets its knowledge, sets the *timerMonitor* and then waits the replies for the maximum time needed to send and receive a message (2δ).

When a node p_i receives the ping, it replies with a "pong" message and attaches its level and the set *joined* of nodes that it knew have joined in the last graph lifetime.

When the supervisor receives the pong message, it updates its knowledge and when the *timerMonitor* expires it computes the new graph containing all the nodes that have replied to the ping and the nodes that have completed their join before the reconfiguration starts. The new graph is computed by means of the *compute_graph*() function that has as parameters the set of nodes to be connected and the degree of connectivity k and returns the new graph. This function can build whichever type of k-connected graph because the graph topology is irrelevant for our algorithm since it uses this function as black box. Once the new graph is computed, the supervisor sends it to all the nodes it knows and then sets again the *timerGraph*.

When a node p_i receives the graph, it updates its data structures; if p_i was not active, it changes its state and from now on it becomes effectively part of the ON and it starts the (transitive) monitoring of the supervisor.

3.2.4 Supervisor Election Procedures

The supervisor can crash and then it is necessary to elect a new one. The monitoring procedure uses the particular structure of the knowledge and avoids all the nodes to ping the supervisor delegating this task only to the nodes at the supervisor level or at the subsequent level. Since every node can crash this local monitoring is repeated for all the levels of the *myKnowledge* structure and each node monitors transitively the supervisor by means of a local monitoring.

In Figure 4 it is shown the monitoring procedure.

The monitor procedure is activated as soon as a node becomes active and it is executed periodically. Let us consider a process p_i, it sends a heartbeat message to all the nodes contained in the parents and candidates lists and waits for their replies setting a timer, namely *timerElection*.

When a node receives the heartbeat message, it sends back a heartbeat reply and, if it did not know the sender of the message, it adds the sender to the list of candidates.

Receiving the heartbeat reply, every process updates its data structure according to the level of the sender node.

When the *timerElection* expires, the election procedure starts as shown in Figure 5.

The election procedure selects a new supervisor when the crash of the current one is detected. The crash of the supervisor is detected by the nodes whose level is the supervisor one and from nodes at the first level not empty following the supervisor level; when these nodes do not receive any reply

```
monitorSupervisor()
 1 while (active every 2δ) do
 2   for each (p ∈ parents ∪ candidates) do
 3     send ("HB_Req, myLevel'') to p
 4   parents = ∅
 5   candidates = ∅
 6   set timerElection = 2δ
```

(a)

```
 1 when (receive ("HB_Req, level'') from p_i) do
 2   send ("HB_Rep'', myLevel) to p_i
 3   if (p_i ∉ knowledge[level])
 4       then joined = joined ∪ p_i
 5            knowledge[level] = knowldge[level] ∪ p_i
 6            if (level == myLevel)
 7                then candidates = candidates ∪ p_i
```

(b)

```
 1 when (receive ("HB_Rep'', level) from p_i) do
 2   if (level < myLevel)
 3       then parents = parents ∪ p_i
 4       else candidates = candidates ∪ p_i
```

(c)

Figure 4: Monitoring Supervisor Procedure

```
when (timerElection expired)
 1 if (|parents == 0|)
 2     then supervisor = min(candidates)
 3 if (supervisor = myId)
 4     then set timerGraph = ϵ
 5     else knowledge[myLevel] = candidates
 6          knowledge[parentsLevel] = parents
 7          parentsLevel = max_i{i < myLevel∧
 8          knowledge[i] ≠ ∅}
```

(a)

Figure 5: Election Procedure

by the "parents" nodes, they know that they are potentially candidates to become supervisor and then decide with a deterministic rule the supervisor. The new supervisor is now ready to build the new graph and starts the reconfiguration procedure.

3.3 Correctness and Guarantees of the Algorithm

In this section we show that our algorithm satisfies eventual connectivity.

THEOREM 1. *The reconfiguration procedure satisfies Eventual Connectivity.*

PROOF. (Sketch) We first show that the reconfiguration procedure maintains connectivity when the supervisor does not crash and then we show that it works even with fault supervisor.

At the beginning there is only one node p_i that becomes supervisor and activates the reconfiguration thread. p_i is the only node active in the ON and the graph G_0 of the ON is composed only by p_i; moreover, both the knowledge and the election information are composed only by p_i. Since the supervisor is the only active node, the bootstrap service returns always its identifier to the incoming nodes. When the reconfiguration procedure starts due to the expiration of the *timerGraph*, the supervisor starts to ping all the nodes who it knows about and waits for 2δ for the replies. We may have two cases (i) no node has joined the ON in L_{G_0}; (ii) some nodes have joined the ON in L_{G_0}. In the first case in 2δ p_i will receive only its own reply, due to the synchrony of the system, and will compute the graph containing again only itself. In the second case, the nodes who have joined the ON and are still alive will receive the ping message in δ time due to the synchrony and the perfect link, and then will reply to the supervisor that will receive the pong message after at most δ time. After 2δ from the ping messages, the supervisor will know exactly how many and who are the nodes alive to be connected in the new graph G_1. If there is enough nodes to build the k-connected graph then a graph including all the nodes is computed otherwise G_1 will include only p_i.

Let us consider a graph G_i with more than one node; when the *timerGraph* expires the supervisor repeats the procedure described above. Since a k-connected graph is able to tolerate $k-1$ failures, the graph lifetime is $L_G = (k-1)/f_r$ and the *timerGraph* expires after $((k-1)/f_r) - 4\delta$ we have that the connectivity is maintained when the reconfiguration starts and it is guaranteed for 4δ more time. To verify which nodes are still alive between the known ones, the supervisor uses 2δ times and to spread the new graph the supervisor uses one more δ time then when the new graph is installed, the old one is still connected. Moreover the new graph is k-connected again and then connectivity is preserved.

Consider now the case where the supervisor can fail. Let i be the level of the supervisor. Due to the *monitorSupervisor*() procedure, every 2δ time nodes at level i and $i+1$ send a heartbeat message to the supervisor. Let us suppose that at some t the supervisor crashes. If there are no other nodes

at level i, when the *timerElection* expires, nodes at level $i+1$ have the set *parents* empty and then will execute the line 2 of Figure 5. Since the rule used to select the supervisor is deterministic, all the nodes will recognize the same new supervisor and the new supervisor knows that it is the new one. Similarly, a new supervisor can be selected deterministically even if other nodes are still at level i; the only difference is that now the supervisor will be chosen between the nodes still alive at level i. The new supervisor starts now the reconfiguration procedure and then the new graph could be installed.

Due to the core assumption, we have that inside the system there exist stable nodes and eventually one of them will be selected to become supervisor and then it never crashes. When a stable node is selected, we return to the case described above where the supervisor does not crash and then from that point connectivity is guaranteed forever.

□

4. CONCLUSIONS

This paper presented a new core-based reconfiguration algorithm able to build an overlay network highly reliable. In terms of deterministic guarantees, the overlay is able to let queries crossing the network be ever satisfied from some point of time on. This limits the possible continual lost of reliability current reconfiguration approaches suffer from. In fact, our approach can lose queries only for a finite time. This finite-time unreliability comes from the fact that members of the core are a-priori unknown to any process joining the network. Nevertheless, all processes will be eventually able to select a core member that will carry out all the reconfigurations. By assuming known an upper-bound on the node failure-rate, successive reconfigurations done by the same supervisor will take effect before the overlay degrades its functionalities.

5. REFERENCES

[1] A. Allavena, A. Demers, and J. E. Hopcroft. Correctness of a gossip based membership protocol. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 292–301, New York, NY, USA, 2005. ACM Press.

[2] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *In Proceedings of The International Conference on Dependable Systems and Networks (DSN '01), July 2001.*, 2001.

[3] A. J. Ganesh, A. Kermarrec, and L. Massoulié. Peer-to-Peer Membership Management for Gossip-Based Protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[4] P. B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. *ACM SIGCOMM Computer Communication Review*, 36(4):147–158, 2006.

[5] V. Gramoli, A. Kermarrec, A. Mostefaouii, M. Raynal, and B. Sericola. Core persistence in peer to peer systems: Relating size to lifetime. In *On The Move International Workshop on Reliability in Decentralized Distributed systems(OTMÕ06), October 2006.*, 2006.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[7] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[8] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[9] S. Voulgaris, D. Gavidia, and M. Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.