Concurrent Connectivity Maintenance with Infinitely Many Processes
Ph.D. Thesis Plan

Sara Tucci Piergiovanni

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY AT "LA SAPIENZA" UNIVERSITY ROME, ITALY OCTOBER 2005
# Table of Contents

1 The Connectivity Maintenance Problem   6  
1.1 Computational Model and Basic Definitions . . . . . . . . . . . . . . . . . . 6  
1.2 Overlay Connectivity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  
1.3 Connectivity Maintenance Specifications . . . . . . . . . . . . . . . . . . . 8  
1.3.1 Overlay Connectivity . . . . . . . . . . . . . . . . . . . . . . . . . . . 8  
1.3.2 Overlay Progress . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11  
1.3.3 Impossibility Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  

2 Connectivity Maintenance Protocols in Absence of Failures 16  
2.1 Protocols implementing Active Weak Connectivity . . . . . . . . . . . . . . . 17  
2.1.1 The Basic Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
2.1.2 Proof of the Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
2.1.3 Protocol Variants Handling False Children . . . . . . . . . . . . . . . . 26  
2.1.4 Maintenance of a Topology . . . . . . . . . . . . . . . . . . . . . . . . 34  
2.2 A Protocol Implementing Active Strong Connectivity . . . . . . . . . . . . . 35  
2.2.1 Proof of the Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39  
2.3 Connectivity Maintenance and Broadcast Problems . . . . . . . . . . . . . . 41  

3 Connectivity Maintenance Protocols Handling Failures 43  
3.1 Key Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43  
3.1.1 A Failure in a Continuous Evolving System . . . . . . . . . . . . . . . . 43  
3.1.2 Limitations of Maintaining Connectivity . . . . . . . . . . . . . . . . . 44  
3.1.3 The Meaning of a Leave . . . . . . . . . . . . . . . . . . . . . . . . . . 44  
3.2 Implementing Eventual Strong and Weak Connectivity . . . . . . . . . . . . . 45  
3.2.1 A Basic Protocol Variant Handling Failures . . . . . . . . . . . . . . . . 45  
3.3 Techniques to Keep Scalability . . . . . . . . . . . . . . . . . . . . . . . . . 50  
3.3.1 A Basic Protocol Variant maintaining Maximum Degree $D$ . . . . . . 51
4 Performance Evaluation  
4.1 The SCAMP Probabilistic Protocol[21] .......................... 55
4.2 The DET Protocol .................................................. 57
4.3 Simulations .............................................................. 58
  4.3.1 Simulation Framework ........................................... 58
  4.3.2 Experimental Results with no Bootstrapping ................. 60
  4.3.3 The Impact of Bootstrapping in SCAMP ....................... 64

5 Related Work  
5.1 Group Membership for Group Communications .................. 67
5.2 Maintenance of Unstructured Overlays ........................... 69
5.3 Maintenance of Structured Overlays ............................... 71

Conclusions and Future Work  

Bibliography
Introduction

Peer-to-peer (P2P) systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources (content, CPU cycles, storage, bandwidth). P2P systems are currently subject of a vast research effort in several fields (informative systems, parallel computing, database management, distributed systems, communication networks). The reasons behind their increasing popularity comes, to a large extent, from their ability to scale, and self-organize in the presence of highly transient population and failures without requiring the intermediation, or support, of a global centralized server or authority [6].

The most prominent application areas of P2P systems are communication and collaboration (e.g. instant messages, chat), and content distribution. Content distribution includes systems and infrastructures designed for data sharing among users [2, 1, 26]. These application areas rely on overlay-based solutions such as, application-level multicast and content-based location and routing.

An overlay network is formed on top of – and generally independently from – the underlying physical computer network, by the peers (nodes) of the P2P system. The arcs (or edges) connecting nodes, reflect some logical relationship among nodes defined at application level.

When solutions to information dissemination and data location/retrieval over P2P systems have been devised, the first P2P system characteristic taken into account has been the large scale. For instance, many application-level multicast protocols [28, 11] rely on a overlay network showing a tree topology, defined by node’s neighbor variables, to span over the entire network optimizing the message overhead. Other examples come from routing protocols like Chord [27], CAN [47], Tapestry [52], Pastry [46], that induce an overlay among nodes with a rich structure for an efficient data location/retrieval. For instance, in Chord [27] physical nodes are mapped in a logical ring and data retrieval is performed with $O(\log N)$ hops in a system of $N$ nodes. These overlays are often called structured overlays. Other efforts are devoted to the construction of overlays with randomized topologies showing some desired properties like low-diameter, high connectivity, constant-node degree [41]. These overlays are often called unstructured overlays. To this class, also belong overlays built (by a gossip-based protocol) to support gossip-based dissemination [18], [3], [21].

The typical approach to the design of these overlays is the following [15]. Given a problem, e.g. information diffusion, a proper graph showing some desired properties is selected (e.g. a random graph, a tree, a ring) – this graph represents the ideal overlay.
A topology is connected if and only if there exists one path between every two nodes of the overlay.
• churn, intended as continuous joining and leaving of processes at arbitrary times; and

• (cumulative) failures.

Note that all these three forms of non-determinism bring the overlay away from its ideal topology, disrupting desired properties like connectivity.

Thus, any solution should be positioned in a general framework by explicitly declaring what are the adopted assumptions or control mechanisms to reduce, at some extent, every source of non-determinism. To the best of our knowledge a few works are clearly doing so. One of them is the recent work of Liben-Nowell et al. [15] that assumes for each of the three forms, respectively: (i) perfect failure detection, (ii) knowledge of the churn rate and system size, and (iii) random failures. The solution based on these assumptions assures high connectivity with high probability.

Unfortunately, there is even a little consensus on what are the reasonable assumptions to make. For instance, is it reasonable to have a protocol that assures connectivity only if the estimated churn rate holds? Moreover, what are the effects if the failure detection is not perfect? If some of these conditions occur, is the maintenance protocol able to bring the overlay towards an ideal state again? This little consensus comes from a limited understanding of the evolution of a P2P system [49].

From our perspective, it is necessary to better understand the problem of connectivity maintenance for any ideal topology by (i) taking into account the whole non-determinism affecting P2P systems and (ii) without limiting the non-determinism (as much as possible) by assumptions. It is worth to note that, in particular, the issue of churn has received little attention in literature. At the best of our knowledge, the effects of the non-determinism generated by churn have not still been investigated.

**Contribution.** The focus of this thesis is the study of dynamic P2P systems connectivity. This problem is formally and empirically analyzed, resulting in the following contributions.

The first contribution of this thesis is defining a computational model abstracting a P2P overlay. The two key assumptions abstracting a P2P overlay are (1) the assumption of an “infinite arrival model” and (2) the assumption of “unbounded level of concurrency” [20, 38]. Specifically, assuming unbounded concurrency means that the (finite) number of processes simultaneously joining/leaving the overlay is arbitrary and unknown. This also implies that the number of processes in the overlay is unknown. The infinite arrival model abstracts a continuously running overlay to which infinite processes eventually will belong. This also implies that the number of cumulative failures tend to infinite.

The second contribution consists in a formal definition of the overlay connectivity maintenance problem. The overlay connectivity maintenance problem is formally stated through a hierarchy of four specifications. The strongest one requires connectivity at any time and immediately enables communication among nodes. The weaker one requires connectivity at any time and immediately enables communication among nodes. The weaker one requires that eventually the overlay never partitions. A possible communication among nodes of the overlay,
however, is not immediately enabled since it is not assured the symmetry of the communication for each pair of nodes. Other properties are specified to guarantee the overlay progress, i.e. to avoid trivial maintenance protocols that systematically prevent joins to the overlay. By defining overlay connectivity properties, we will formally delimit boundaries of its solvability under the model specified. In particular some immediate impossibility results are proved.

The third contribution is the implementation of overlay connectivity maintenance under the model specified in absence of failures. The absence of failures has allowed to study the problem only in presence of churn (joins and voluntary leaves) and asynchrony. The impact of these two forms of non-determinism has revealed some interesting results. Specifically, it is shown that (i) it is possible to maintain connectivity at any time, but the level of asynchrony has an impact on the memory required at each node, and (ii) it is possible to maintain a specified topology (e.g. a tree) only during stabilization periods of the overlay in terms of joins/leaves.

The fourth contribution is the implementation of protocols managing connectivity considering also the presence of failures. To cope with the presence of cumulative failures, the presented connectivity maintenance protocols has been equipped with a restoring mechanism in order to repair the overlay at any time. This study has pointed-out several interesting results. First, connectivity cannot be assured at any time, but it is guaranteed only during stability periods of the overlay in terms of joins/leaves. Asynchrony impacts on the scalability of the topology: these protocols shrink the topology to a small set of hubs (to a star in the worst case) during asynchrony periods of the underlying network. In particular, false suspicions bring nodes to restore by re-connecting to few nodes still in the overlay, i.e. non-restoring nodes. To avoid that asynchrony influences connectivity, i.e. that false suspicions compromise connectivity, infinite memory is required.

The fifth contribution is a performance evaluation that points out a trade-off between scalability of the overlay and its level of connectivity. In particular, we have compared an extremely scalable peer-to-peer maintenance protocol like SCAMP [21] that maintains connectivity with high probability and Det, i.e. one of the protocols proposed in this dissertation. We have evaluated the effects of churn on both protocols. Not surprisingly, under high churn rate DET loses scalability (during high churn rates the overlay shrinks to a star) in order to guarantee connectivity, while Scamp loses in connectivity maintaining high scalability (in particular constant node degree).

The contributions listed above and basic ideas from which this work has started are partially contained in the following papers: [45, 43, 44, 8].

**Thesis organization.** In Chapter 1, the model (Section 1.1) and the connectivity maintenance problem are formally specified (Section 1.3). In Section 1.3.3 some impossibility results are shown.
In Chapter 2, a set of protocols guaranteeing connectivity in absence of failures are presented. In particular, Section 2.1 presents protocols assuring that the overlay never partitions. At any time the overlay is a connected directed acyclic graph: the communication is transitive but not symmetric for each pair of nodes. Symmetry is, however, assured during the overlay stabilization periods. In stabilization periods is also assured the maintenance of a tree topology. In Section 2.2 a protocol assuring that the overlay is a connected undirected graph at any time is shown. In this case symmetry is always assured. All protocols are presented along their correctness proof. In Section 2.3 relationships between connectivity specifications and broadcast communication are discussed.

In Chapter 3, protocols implementing connectivity under failures are presented. In particular, in Section 3.1 some key concepts justifying our design choices are discussed. Section 3.2 presents a protocol assuring connectivity only during stabilization periods of the overlay. Interestingly, as for the liveness properties of group membership [13], assuring connectivity is conditional on the overlay stabilization (see Chapter 5 for further details).

In Chapter 4 an experimental analysis conducted to evaluate the impact of churn on Scamp and Det is presented. In Section 4.1 Scamp [21] is briefly described. Section 4.2 explains some technicalities used to simulate Det. In Section 4.3 experimental results are shown.

Chapter 5 discusses the related work, the last chapter draws the conclusions and discusses the future work.
Chapter 1

The Connectivity Maintenance Problem

In this chapter the Connectivity Maintenance problem is formally specified. Different specifications are presented and the solvability of the problem will be studied in an distributed system composed by an infinite number of processes under unbounded concurrency.

1.1 Computational Model and Basic Definitions

The system is composed by an infinite set of processes $\Pi$, uniquely identified. We will denote processes by $i, j, k$. Processes communicate only by exchanging messages through point-to-point reliable channels.

The system is assumed asynchronous, i.e., there is no bound on the transmission delay of a message and any internal processing at a process takes an arbitrary but finite time.

A process may be correct or faulty. A correct process never fails. A faulty process fails by crashing.

To simplify the description without losing generality, we assume the existence of a fictional global clock, whose output is the set of positive integers denoted by $T$.

**Process State.** Each process $i \in \Pi$ has a finite set $i.X$ of variables $x \in \Pi \cup \{\text{nil}\}$, where $\text{nil}$ is a special process that does not belong to $\Pi$. There is no communication channel from/to $\text{nil}$. $i.x$ denotes a neighbor variable $x$ of process $i$. By definition the $\text{nil}$ process does not have any neighbor variable. The state of a process $i$ is represented by $i.X$. Each neighbor variable $i.x$ initially assumes a $\text{nil}$ value.

**Overlay Definition.** At any time, the global state constituted by the set of processes and their neighbor variables define the overlay. Formally,

**Definition 1.1.1 (Overlay $O(t)$).** The overlay $O(t)$ is the set constituted by each pair $\langle i, i.X \rangle : \exists i.x \in i.X \neq \text{nil}$ at time $t$. 
Any process $i : \langle i, i.X \rangle \in O(t)$ is called a vertex of the overlay $O(t)$.

**Process Actions.** Each process $i$ is characterized by the following actions: $\text{init}_i$, $\text{join}_i$, $\text{leave}_i$, $\text{stop}_i$.

The $\text{init}_i$ action is an output action coming from the outside world (by an application). The effect of the $\text{init}_i$ action is the initialization of the neighbor variables. The $\text{join}_i$ action is an internal action. The effect of the $\text{join}_i$ action is the assignment of a non-nil value to some neighbor variable of $i$, from a state in which all neighbor variables have a nil value, i.e. the $\text{join}_i$ action determines the addition of the process $i$ to the overlay.

The $\text{leave}_i$ action is an internal action. The effect of the $\text{leave}_i$ action is the assignment of nil values to each neighbor variable of $i$, from a state in which at least one neighbor variable has a non-nil value, i.e. the $\text{leave}_i$ action determines the deletion of the process $i$ from the overlay.

The $\text{stop}_i$ action is an input action that models a crash of the process. The effect of this action is the same for the $\text{leave}_i$ action, i.e. the $\text{stop}_i$ action determines the deletion of the process $i$ from the overlay. After $\text{stop}_i$ no other actions occur.

**Concurrency.** The admitted level of concurrency is finite and unbounded [38, 20]. The number of processes that may be concurrently added/deleted to/from the overlay at any time is finite, arbitrary and unknown. This implies that at any time $t$ the number of pairs $\langle i, i.X \rangle \in O(t)$ is finite but can grow without bound.

We assume an infinite arrival model [38, 20], i.e., the number of processes that globally will add to the overlay (for any infinite system execution) may be infinite. However, once a vertex $i$ is deleted from the overlay (by $\text{leave}_i$ or $\text{stop}_i$), it is deleted forever. This assumption is not restrictive as a process can become a vertex with a different identifier an infinite number of times (by infinite arrival model) thanks to the infinite nature of $\Pi$. By infinite arrival model, the set of vertices in the overlay may change infinitely often. This implies that there is no assumption on eventual stable overlays, i.e. there is no point of time $t$ after which $O(t) = O(t')$ for each $t' > t$.

A vertex may be stable or transient. A stable vertex, once added to the overlay, belongs to the overlay forever. A transient process belongs to the overlay only a finite interval of time.

Formally,

**Definition 1.1.2 (Stable Vertex).** $i$ is a stable vertex iff $\exists t : \langle i, i.X \rangle \in O(t') \forall t' \geq t$.

**Definition 1.1.3 (Transient Vertex).** $i$ is a transient vertex iff $\exists [t, t'] : \forall t'' \in [t, t'], \langle i, i.X \rangle \in O(t'') \land t' < \infty$.

### 1.2 Overlay Connectivity

The overlay $O(t)$ is connected if and only if for any pair of vertices there is a path of neighbor variables connecting them.
Note that given two pairs \( \langle i, i.X \rangle, \langle j, j.X \rangle ∈ O(t) \), \( i ∈ j.X \) does not imply that \( j ∈ i.X \). For this reason at any time the overlay may be viewed as a digraph with an arc \((i, j)\) if \( i ∈ j.X \). With some abuse of terminology we will refer to the overlay either as a global state or as the graph defined by its neighbor variables.

A directed path from \( i \) to \( j \) at time \( t \) is denoted as \( i →_t j \) and is defined as follows:

**Definition 1.2.1 (Directed Path).** For each \( i, j : \langle i, i.X \rangle, \langle j, j.X \rangle ∈ O(t) \), then \( i →_t j \) iff at time \( t \) one of the following conditions holds:

\[
\exists i.x ∈ i.X : i.x = j
\]

or

\[
\exists k ∈ \Pi : (i →_t k) ∧ (k →_t j)
\]

Two vertices are weak connected if there exists a directed path in between, formally:

**Definition 1.2.2 (Weak Connection between Two Vertices).** Any pair \( i, j : \langle i, i.X \rangle, \langle j, j.X \rangle ∈ O(t) \) (\( i \) may be equal to \( j \)) are weak connected, iff \( i →_t j \lor j →_t i \).

Two processes are strong connected if there exist two directed paths (one in each direction) in between, formally:

**Definition 1.2.3 (Strong Connection between Two Vertices).** Any pair \( i, j : \langle i, i.X \rangle, \langle j, j.X \rangle ∈ O(t) \) (\( i \) may be equal to \( j \)) are strong connected, iff \( i →_t j \land j →_t i \).

### 1.3 Connectivity Maintenance Specifications

Any connectivity maintenance protocol dynamically manages the overlay in order to add and to delete vertices while keeping the overlay connected. In the following we give (i) a set of properties which predicate on the state of the overlay and (ii) a set of properties that guarantees the progress of the overlay.

#### 1.3.1 Overlay Connectivity

Firstly, we specify two properties that guarantee overlay connectivity at any time. Then, we specify other two properties that guarantee overlay connectivity eventually.

The first one called Active Strong connectivity (AS) guarantees that any pair of vertices in the overlay are strongly connected at any time. The second one called Active Weak connectivity (AW) guarantees that any pair of vertices in the overlay are weak connected at any time. A protocol assuring at least active weak connectivity avoids partitioning. A protocol assuring also active strong connectivity avoids partitions and guarantees at each vertex the possibility of communicating at any time with other overlay vertices.
The third and the fourth, called Eventual Weak/Strong Connectivity \((EW, ES)\), guarantee that any pair of vertices in the overlay are weak/strong connected only from an arbitrary point of time onwards. Thus, a protocol assuring only eventual weak or strong connectivity may not avoid partitioning. It avoids partitions from an arbitrary time onwards. Formally,

**Property 1.3.1 (Active Strong Connectivity \((AS)\)).** At any time \(t\),
\[
\forall i, j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t) \Rightarrow (i \rightarrow_t j) \land (j \rightarrow_t i)
\]

**Property 1.3.2 (Active Weak Connectivity \((AW)\)).** At any time \(t\),
\[
\forall i, j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t) \Rightarrow (i \rightarrow_t j) \lor (j \rightarrow_t i)
\]

**Property 1.3.3 (Eventual Strong Connectivity \((ES)\)).** \(\exists t : \forall t' \geq t,\)
\[
\forall i, j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t') \Rightarrow (i \rightarrow_{v'} j) \land (j \rightarrow_{v'} i)
\]

**Property 1.3.4 (Eventual Weak Connectivity \((EW)\)).** \(\exists t : \forall t' \geq t,\)
\[
\forall i, j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t') \Rightarrow (i \rightarrow_{v'} j) \lor (j \rightarrow_{v'} i)
\]

**Implementation Issues.** The connectivity problem is a real challenge in a P2P system because of the presence (i) asynchrony, (ii) the continuous arrival/departure of correct processes at arbitrary times, and (iii) faulty processes.

To point out what implementation issues arise under this model, in the following we will first discuss implementation issues in this two simplified models: (i) a model only affected by asynchrony and continuous arrival/departure of correct processes at arbitrary times, and (ii) a model only affected by asynchrony and failures. This allows to highlight some characterizing implementation issue when facing either churn or failures. Then, implementation issues combing asynchrony, churn and failures are discussed.

**Asynchrony and continuous arrival/departure of correct processes at arbitrary times.** To assure Active connectivity \((AW, AS)\), the continuous arrival/departure of correct processes forces a connectivity protocol to continuously update neighbor variables each time new processes join/leave the overlay. In particular, any protocol should follow an active approach \([34]\): for each join or leave it should execute a piece of code before the actual addition/deletion of the element to/from the overlay. It is worth to note that, while overlay changes occur, the neighbor variables have to be updated, i.e. the neighbor variables cannot report a definitive value.

To assure Eventual connectivity \((EW, ES)\), the approach to follow is the same. In fact, if overlay changes never subside, assuring eventual connectivity boils down to assure continuous connectivity.

**Process Failures.** In this case we are considering a classical static distributed system model with a fixed number of processes \(n\). Generally, process failures could be faced in two ways:
(1) exploiting a certain degree of redundancy (a same process in different neighbor variables) or/and (ii) relying on some detection/recovery mechanism when failures occur.

To assure Active connectivity (AW, AS) redundancy is needed since a detection/recovery mechanism could not be instantaneous. The more the redundancy built by the protocol, the more the number of tolerated failures. The level of redundancy defines a threshold of tolerated failures. While the threshold holds, AW, AS are assured. Upon the threshold exceeding, the protocol does not work anymore. It is worth to point out that to implement Active connectivity a detection/recovery mechanism has to be used carefully. Active connectivity obliges to have redundancy. The detection/recovery mechanism could help to restore the level of redundancy when one failure occurs to tolerate one more failure. However, the not instantaneous detection of a failure may prevent of obtaining this effect, i.e. the threshold of tolerated failures does not increase adopting a detection/recovery mechanism. Someone may argue that a detection mechanism could be anyhow useful to remove all faulty processes from neighbor variables. If the model, however, is not augmented with perfect failure detectors any false suspicion may bring to accidentally decrease redundancy. Resuming, the threshold of failures is defined by the level of redundancy, a recovery mechanism does not deterministically help and a detection mechanism has to be perfect.

To assure Eventual connectivity (EW, ES), then redundancy may be even avoided (e.g. redundancy has a not negligible cost an application-level multicast running over the overlay). By relying on some detection/recovery mechanism, the overlay could be restored each time it breaks. In each run, after all failures have occurred, this mechanism would allow to restore the overlay once and for all. To accomplish this task, however, the model has to be augmented with a failure detector assuring completeness to eventually restore a broken overlay. Note that implementing Eventual connectivity with a detection/recovery mechanism allows to tolerate an unbounded number of failures.

Continuous arrival/departure of (faulty) processes at arbitrary points of times.

To guarantee Active connectivity (AS and/or AW), a protocol could follow an active approach to manage joins and/or leaves. To cope with failures, the possibility of maintaining a proper degree of redundancy in the overlay, assuming that a certain threshold of failures holds, could be investigated. Two problems, however, arise. The first one is the difficulty of maintaining a proper degree of redundancy at any time despite join/leaves and failures. The second and more important problem regards the utility of such an approach. What does it mean tolerating a certain threshold of failures in a dynamic setting? Actually, in a system in which (i) an unbounded set of vertices may belong to the overlay and (ii) that set may change infinitely often, e.g. it may grow without any bound; tolerating a certain threshold of failures means tolerating a non-deterministic fraction of faulty vertices. For instance, if a protocol tolerates one failure, it deterministically tolerates one failure both with 2 vertices and with one thousand of vertices. Furthermore, as soon as 2 failures occur the specification is violated. Thus, redundancy does not add any sensible value to assure a deterministic property (such an active connectivity) in a continuous evolving system. It is undoubt, however, that redundancy gives the possibility of tolerating more or less failures in a probabilistic way depending on the topology the overlay is able to maintain. In the
following Active connectivity is not implemented under failures.

To assure Eventual connectivity \((ES, EW)\) a detection/recovery mechanism could be pursued as in the static case. Assuring Eventual connectivity means, however, that there exists a point of time after which the connectivity is restored and maintained forever. With continuous replacement of vertices, this condition holds only if (i) joins and leaves eventually subside and (ii) the model is augmented with a fault-detection mechanism enjoying completeness (as in the static case). It is worth to note that, to assure Eventual connectivity, the leave of a process may also be handled in a passive way, i.e., each leave is assimilated to a crash (stop). Then a passive approach boils down to restore (with a repair protocol) the connectivity whenever connectivity is broken because of leaves and failures. In this case the difference between an active or a passive approach lies in the possibility of not losing connectivity for the active leave. Moreover, the active leave performs this proactive restoring locally. In many situations leaves occur more frequently than failures. In such situations, with a lazy fault-detection (large time-outs), an active approach may lead to improve performance (e.g. saving background traffic) and the level of connectivity over the time.

In this dissertation an active approach to the implementation of active and eventual connectivity is investigated. Specifically, an active approach is the adopted mechanism in order to reduce the non-determinism coming from churn.

1.3.2 Overlay Progress

The properties listed in Section 1.3.1 safeguard the connectivity of an overlay that may change infinitely often due to the level of concurrency and the infinite arrival model assumed. However, a trivial (and bogus) connectivity maintenance protocol may systematically avoid the addition of any element in the overlay. Then, such a protocol would strengthen the model, by reducing it to a static system.

To avoid a trivial protocol that systematically prevent any addition, properties on the progress of the overlay must be formally stated.

Before introducing such properties let us introduce different ways in which the overlay can be initialized.

Overlay Initialization

Generally, we can consider that the overlay at time \(t_0\) \((t_0\) is the smallest time in which one process \(i \in \Pi\) executes its \(\text{init}_i\) action\) may be initialized in the following ways:

- \(|O(t_0)| = 0\).
- \(|O(t_0)| = 1\). In this case we are supposing that the overlay upon its initialization contains one element \(\langle i, i.X \rangle : |i.X| = 1 \land i \in i.X\). It means that a particular process \(i\) called the initiator has a special \(\text{init}_i\) action, different from other \(\text{init}_j\) actions. The \(\text{init}_i\) initializes \(i.X\) neighbor variables in the following way: \(\exists x \in i.X : x = i \land \forall y \in i.X : y \neq i, y = \text{nil}\).
\(1 < |O(t_0)| < \infty\). In this case we are supposing that the overlay upon its initialization contains several elements. The overlay state, however, should have a safe configuration even at \(t_0\) to satisfy AW and AS. Thus, it does not make any sense to consider this type of initialization, the most general cases are the first two in which the connectivity protocol starts with an overlay empty or containing at most one element.

In the following, we will consider overlays starting at \(t_0\) with \(|O(t_0)| = 0\) or \(|O(t_0)| = 1\).

**Progress Properties**

Two different progress properties are formally stated: **Global Progress** and **No-Lockout**.

**Property 1.3.5 (Global Progress).** If \(\exists i \in \Pi \) that executes the \(\text{join}_i\) action at time \(t\), then \(\exists t', j : t' > t \land \langle j, j.X \rangle \in O(t') \land \langle j, j.X \rangle \notin O(t)\)

This property does not guarantee that the access to the overlay is granted to all processes executing the connectivity protocol, i.e. it allows a process \(j\) to be repeatedly granted to access the overlay, while another process \(i\) trying to gain the access is forever prevented from doing so. The following stronger progress property avoids this scenario.

**Property 1.3.6 (No-Lockout).** \(\forall i \in \Pi \) that executes the \(\text{join}_i\) action, \(\exists t : \langle i, i.X \rangle \in O(t)\).

In the following we will see that **Global Progress** and **No-Lockout** can be guaranteed only if some system conditions hold. In practice, there is an impossibility to guarantee **Global Progress** if the initiator does not exist (\(|O(t_0)| = 0\)). Then there is an impossibility to guarantee **No-Lockout** if the overlay is affected by continuous and total replacement of its vertices.

### 1.3.3 Impossibility Results

Before formally stating impossibility results, let us introduce the way in which processes enter the overlay.

**The Overlay Access Problem**

By definition, any process, to be added in the overlay, must set one (or more) of its own neighbor variables to a non-nil value. How to set these variables depends on which specification the protocol is implementing. In any case, to assign some process identifier to a neighbor variable, the need of getting this identifier arises. In other words, the process must have a way to know some vertex in the overlay (a-priori knowledge is not assumed).

For this reason, we assume that any process that wishes to join the overlay can invoke a \(\text{contact}()\) function. The \(\text{contact}()\) function returns a set of processes called \(\text{contacts}\). Ideally, this function should be as simpler as possible, but to make the problem treatable we assume that this function returns an arbitrary set of vertices. Due to concurrency, when a process obtains the \(\text{contacts}\) set, this set could either comprise some process no more belonging to the overlay or can miss some current vertex. This unavoidable fact leads to some impossibility results.


**Impossibility Results**

To abstract our problem and to show impossibility results, let us consider a very powerful contact function that allows processes to get the complete snapshot of the overlay at the time they invoke the function. Then, on the basis of the information got from the contact function, the process enters the overlay invoking an `update()` function. Once the update function returns the process has been added in the overlay.

To avoid solving our problem with the contact function, we also suppose that the invocation of the contact function is only intended to solve the bootstrap, i.e., it can be invoked at most once and before the invocation of the `update()` function.

Fig. 1.1 resumes this simple protocol. The variable $O$ represents the overlay. $O[0...i...\infty]$ is an array of infinite elements $i.X$ indexed by $i$.

Any process $i$ can read this array through the `contact()` function. `contact()` returns in `contacts` the finite number (by finite concurrency) of non-nil $O[i]$ values in $O[]$. At this point $i$ executes an `update_i()` on $O[]$. The update consists in writing each $O[j]$ component corresponding to each process in `contacts` and the $O[i]$ component as defined at line 8.

To the aim of proving impossibility results, the access to $O[]$ can be assumed atomic. Impossibilities arise from the non-atomic execution of the read (`contact()`) and the subsequent write (`update()`) on $O[]$.

In particular, the interleaving between reads/writes immediately implies the impossibility of maintaining a clique at any time:

**Theorem 1.3.7.** If there exists a time $t$: $\forall (i, i.X) \in O(t)$, $i.X$ variables define a clique, then there exists a time $t' > t$ in which the graph defined by each $(i, i.X) \in O(t')$ it is not a clique.

*Proof.* Let us suppose by contradiction that for all $t' > t$, the graph defined by $(i, i.X) \in O(t')$ is a (undirected) clique. This implies that for each $t' > t$, $\forall (i, i.X), (j, j.X) \in O(t')$, $\Rightarrow j \in i.X \land i \in j.X$.

Let us consider any run $R$ in which only two processes $i, j \in \Pi$ add to the overlay after time $t$. In particular, they invokes the `contact()` function at the time, respectively, $t'_i, t'_j > t$ and the `update()` function at time, respectively, $t''_i, t''_j > \max(t'_i, t'_j)$. The `contact()` function returns all processes in $O(t)$ (not comprising $i$ and $j$). Then at time $t' = \max(t''_i, t''_j)$ the overlay $O(t')$ contains $(i, i.X)$ and $(j, j.X)$ but $i \notin j.X$ and $j \notin i.X$ contradicting the initial hypothesis.

The above impossibility forces the $i$’s leave to update its neighbors before setting them to nil.

Any $i$’s leave (Fig. 1.1, line 9) is modelled through one atomic statement such that (i) for each $j \in O[i]$, then $O[j] = O[j] \cup O[i]$ and (ii) $O[i] = \text{nil}$.

**Theorem 1.3.8.** If there exists a time $t$: $|O(t)| = 0$, $AW$, $AS$, $EW$, $ES$ cannot be solved.

---

1For the purpose of the proves, it is not necessary to remove $i$ from the neighbor variables of each $j$. 
JOIN
1 var O[.]: array[0,∞] of i.X;
2 init O[nil,...,nil];
3 begin
4 contacts := contact();
5 update(contacts ∪ {i});
6 end

CONTACT()
7 return A : ∀O[i] : O[i] ≠ nil ⇒ i ∈ A;

UPDATE[A]
8 O[i] = A; ∀j ∈ A ∧ O[j] ≠ nil ⇒ O[j] = O[j] ∪ {i}; % Check of nil values because of possible concurrent leaves %

LEAVE
9 ∀j ∈ O[i], O[j] = O[j] ∪ O[i]; O[i] = nil;

Figure 1.1: A Connectivity Protocol at i

Proof. Let us consider any run R in which only two processes i,j ∈ Π execute the protocol. In particular, they invokes the contact() function at time, respectively, $t_i^w,t_j^w > t$ and the update() function at time, respectively, $t_i^w,t_j^w > max(t_i^w,t_j^w)$. This implies that the contact() function returns an empty set. Then at time $t'' = max(t_i^w,t_j^w)$ the variable O contains the following two elements: $O[i] = \{i\}$ and $O[j] = \{j\}$, i.e. a time $t''$, $O(t'')$ has only two pairs $\langle i,\{i\} \rangle$ and $\langle j,\{j\} \rangle$, i.e. $i \not\leftrightarrow_{t''} j$ and $j \not\leftrightarrow_{t''} i$, then $AS$ and $AW$ are violated.

Moreover, let us consider a run $R'$ among the $R$ runs, in which $i,j$ are stable, then even $ES,EW$ are violated. □

To overcome the previous impossibility result, the following assumption should be done:

Assumption 1.3.9. At any time, there exists at least one vertex in the overlay.

Theorem 1.3.10. If there exists a time $t$: ∀(i,i.X) ∈ O(t), i is transient, $AW, AS, EW, ES$ cannot be solved

Proof. Let us consider any run R in which at any time there exists at least one vertex (not fall in the previous impossibility). In particular: (i) $O(t)$ contains only one element $\langle i,i.X \rangle$, only two processes $j,k$ execute the protocol. Let us suppose that they invokes the contact() function at time, respectively, $t_j^w,t_k^w > t$ and the update() function at time, respectively, $t_j^w,t_k^w > max(t_j^w,t_k^w)$. This implies that the contact() function returns $\{i\}$. Let us suppose that $i$ leaves at time $t_j^w < t_i < t_k^w$ .

In this case a time $t_j^w$, $O[i]=\text{nil}$, $O[j] = \{j\}$, $O[k] = \{k\}$, i.e. $O(t_j^w)$ contains $\langle j,j.X = \{j\} \rangle$ and $\langle k,k.X = \{k\} \rangle$, i.e. $j \not\leftrightarrow_{t_j^w} k$ and $k \not\leftrightarrow_{t_j^w} j$, then $AS$ and $AW$ are violated.

Moreover, let us consider a run $R'$ among the $R$ runs, in which $j,k$ are stable, then even $ES,EW$ are violated. □

To overcome the previous impossibility result, the following assumption should hold:
**Assumption 1.3.11.** At any time, there exists at least one stable vertex in the overlay.

The previous impossibilities may not affect connectivity properties but only progress properties if a process adds to the overlay only when connectivity is safeguarded, otherwise it never adds to the overlay.

From the previous results, connectivity is not safeguarded when:

- the contact function returns an empty set

- the update successes even when all overlay components returned by the contact function have nil values, i.e. contacts do not belong to the overlay. Note that the `update()` actually hides a message exchange between `i` and `contacts`. Since `contacts` are vertices, they do not know `i`, then the message exchange starts from `i` that through a message introduces itself to `contacts`. Then, `contacts` update locally their neighbor variables and they send an acknowledgement to `i`. Then, if `i` updates its neighbor variables only upon receiving these acknowledgments, the update does not return if `contacts` have already left the overlay.

If any addition is prevented in the first case, by removing Assumption 1.3.9, *Global Progress* is violated. If any addition is prevented in the second case, by removing Assumption 1.3.11, *No-Lockout* is violated. Practically, to assure *No-Lockout* the overlay should not change for a time long enough to allow the addition of a new vertex.
Chapter 2

Connectivity Maintenance
Protocols in Absence of Failures

In this Section all proposed protocols work assuming an asynchronous system in which no failure occurs. This assumption seemingly makes the connectivity problem a simple problem. However, due to the level of concurrency admitted and the presence of leaves (even though only voluntary leaves), the problem is far from being trivial [34]. Moreover, the presented solution allows to simply introduce several key concepts and exemplifies techniques used for solving the problem in presence of failures.

Since the work presented in [34] is closely related to the work presented in this chapter, differences and common issues will be discussed in Section 2.1.4.

The Basic Idea. The presented protocols arrange the vertices of the overlay in a routed tree topology. The choice of maintaining a tree derives from two facts.

Firstly, our objective is to design a join to the overlay as simple as possible. Our protocols provide the joining process $i$ with a parent (a vertex) and assure that the parent sets $i$ as child. Then, the join completes with the update of only two neighbor variables out of the unbounded number of neighbor variables in the overlay. Moreover, the choice of the parents has not to follow particular rules, i.e. the choice is arbitrary, it is required, however, that the selected parent stays in the overlay enough time to terminate the join (to cope with asynchrony and concurrency). On the contrary, if the join protocol embedded more complexity, as for example, searching a specific position in the overlay, it would not be trivial proving the correctness of this search over an unbounded number of participants under continuous overlay changes. Secondly, the tree is a basic structure used in many applications, e.g. for search algorithms, multicasts, etc. Thus, studying the tree as a basic topology for overlay connectivity should be useful.

The protocols, as said in Section1.3 maintain such a topology by using an active approach, i.e., a piece of code is executed by a leaving/joining process interacting with its neighbors in the topology before the actual addition/deletion to/from the overlay.
2.1 Protocols implementing Active Weak Connectivity

In the following the overlay will be denoted as $T(t)$. Each element of $T(t)$ is composed by a pair $(i, (i.parent, i.children))$ where $i.parent$ and each $x \in i.children$ are neighbor variables.

**Process state.** With the respect to the process state described in Section 1.1, the state is enriched by the variable $s$, the variable $s$ can assume the values \{in, out, joining, leaving\}.

**Process actions.** Each process $i$ performs a `connectivityManagement()` task to maintain the overlay.

We detail the effects of the actions $\text{init}_i$, $\text{join}_i$, $\text{leave}_i$ described in Section 1.1.

The $\text{init}_i$ action has the following effect: (i) it initializes the neighbor variables to nil values \(^1\), (ii) it initializes the $s$ variable to out and (iii) invokes the `connectivityManagement()`.

$\text{join}_i$: sets the $s$ variable to joining.

$\text{leave}_i$: sets the $s$ variable to leaving.

**$T(t)$ initialization.** The overlay is initialized by an initiator $r$, i.e. $T(t_0) = (r, (r.parent = r, \forall x \in r.children, x := \text{nil})) \land r.state = \text{in}$. It means that $r$ is a special process with an $\text{init}_r$ action occurring at time $t_0$ slightly different from the $\text{init}$ of all other processes. Upon the $\text{init}_r$ action, $r$ (i) sets $s := \text{in}$, (ii) $r.parent := r$ and $\forall x \in r.children, x := \text{nil}$. Then, $r$ invokes the `connectivityManagement()`.

**The contact() function.** We assume that the contact function returns an arbitrary vertex $j$. From Assumption 1.3.9 there always exists a vertex in the overlay. This means that the contact() function is able to return a non-nil value at each process. Let us remark that this is a necessary condition to not violate Global Progress.

**No-Lockout** holds only if the contact() function eventually retrieves a stable vertex.

**The connectivityManagement().** The `connectivityManagement()` handles the addition/deletion of the pair $(i, (i.parent, i.children))$ to/from $T(t)$, i.e. it sets and update neighbor variables.

As soon as a process $i$ wishes to join the overlay, it sets its variable $s$ to joining. Any vertex in the tree wishing to leave sets variable $s$ to leaving. The `connectivityManagement` handles the $s$ state transitions: $[\text{joining/in}]$ and $[\text{leaving/out}]$. The completion of the transition $[\text{joining/in}]$ at time $t$ denotes the addition of the pair $(i, (i.parent, i.children))$ to $T(t)$. The completion of the transition $[\text{leaving/out}]$ at time $t'$, denotes the deletion of the pair $(i, (i.parent, i.children))$ from $T(t)$.

\(^1\)The set of neighbor variables is unbounded, i.e. the number of neighbor variables is a-priori-unknown, thus this static initialization models a dynamic initialization of neighbors variables.
With this mechanism, all in processes are vertices of the tree, while all out processes are not vertices of the tree, i.e. each out process \( i \) has \( i.\text{parent} = \text{nil} \land \forall x \in i.\text{children}, x = \text{nil} \).

The state transition \([\text{joining/in}]\) is quite simple to handle, since it does not threaten the connectivity of participants. The critical transition is \([\text{leaving/out}]\), since in the tree topology each leave, but the one executed by a leaf, creates a partition. Then, before deleting the pair \( (i, (i.\text{parent}, i.\text{children})) \) from \( T(t) \), \( T(t) \) has to be updated. In particular one of the following updates should occur:

\[
\forall j \in i.\text{children}, j.\text{parent} := i.\text{parent}
\]

or

\[
\forall k = i.\text{parent}, k.\text{children} := k.\text{children} \cup i.\text{children}
\]

**Handling Concurrency.** In a message passing system, updates are not instantaneous, and any of them involves the exchange of at least one message (the update message). In case of two (or more) concurrent leaving processes \( i, j \) which are adjacent on a same path, the deletion of the pairs \( (i, i.X) \) and \( (j, j.X) \) during the concurrent diffusion of the updates may lead to a partition. To avoid partition, before deleting \( j \), the update related to \( i \) must take effect (or vice versa) to give a notice to \( j \) that its neighbors have been changed. A this point \( j \) can send its update to the new neighbors. In all presented protocols the deletion order is given by the distance from the root. The process closest to the root leaves first.

**The Tree Maintenance.** It is worth to note that the all protocols presented in this section try to maintain a tree at all times where \( \text{parent} \) and \( \text{children} \) variables define the overlay structure. If we consider, however, the two graphs defined respectively by only \( \text{children} \) and the one defined by the only \( \text{parent} \) variables, they are not isomorphic [17]. In particular, the graph defined by \( \text{children} \) variables may be not a (directed) tree since the false \( \text{children} \) problem arises because of asynchrony (see Section 2.1.1).

Moreover, the (directed) graph defined by \( \text{parent} \) variables is a directed tree only while the overlay does not change. Upon an overlay change (a leave), the tree is broken: during the update message exchange the tree defined by \( \text{parent} \) variables is disconnected (see the leave handling in Section 2.1.1).

We will prove that, even if the tree is lost sometimes, our protocols always maintain a connected digraph defined by both \( \text{parent} \) and \( \text{children} \) variables.

### 2.1.1 The Basic Protocol

In Figure 2.3 the Basic Protocol pseudo-code is shown.

From line 3 to 8, the \( \text{connectivityManagement}() \) handles the \([\text{joining/in}]-\)transition, while from 9 to 13 it handles the \([\text{leaving/out}] \) transition. From line 14 to 19 the message handling is shown.
[joining/in]-transition. When the variable \(i.s\) assumes the joining value, the Basic Protocol provides to obtain a vertex \(a\) of the tree through the \(\text{contact()}\) function.

Then \(i\) sends a “JOIN” message to \(a\) (line 5). Upon receiving the “JOIN” message, \(a\) adds \(i\) to its children if its \(a.s = \text{in}\), by setting \(a.children\) to \(i\) and sending an “ACK_JOIN” message back to \(i\) (lines 14,15). Upon receiving the “ACK_JOIN” message, \(i\) sets \(i.parent\) to \(a\) and sets the variable \(i.s = \text{in}\) (line 7-8).

With the mechanism described above, if the joining process establishes its connection at time \(t\) with a vertex \(j\), i.e. \(i \to t j\), then there exists a time \(t' < t\) in which \(j \to t' i\) (see Fig.2.1). This mechanism allows to couple \(i\) to a possible transient parent. As we see in the following, this parent before leaving will give a notice to all its children, \(i\) comprised.

Due to concurrency and asynchrony, the \(a\) vertex given by the \(\text{contact()}\) function could be out before receiving the “JOIN” message from \(i\). This is why after a timeout \(T\), if the joining process has not yet received an “ACK_JOIN” message, it restarts by getting another contact \(a\).

[leaving/out]-transition. When the variable \(i.s\) assumes the leaving value, the protocol allows to update the \(i\)’s neighbors before deleting \(i\) from the overlay. In practice, it ensures that the parent of \(i\) becomes the parent of each child of \(i\).

In particular, the task comprises a sequence of asynchronous basic steps, where a basic step is the sending and the receiving of a “LEAVE” and its “ACK_LEAVE” message, respectively (lines 9-13). The [leaving/out]transition occurs when a basic step succeeds (line 13), \(s = \text{out}\). More in details, during each step, a “LEAVE” message containing the children of \(i\) is sent to update the neighbor variables of the current \(i\)’s parent (line 10).

\(^2\)We assume that a process which has left the overlay (an out process) is not obligated to respond to messages associated with the maintenance of the overlay connectivity. This is in contrast on what assumed in [34], in which even an out process is supposed to respond to overlay connectivity messages.
In the simple case where the $i$’s parent $k$ is in when the “LEAVE” ($i\.children$) message is received (line 16), $k$ updates its children variable, it sends a “NEW\_PARENT” ($k$) message to all new children and it sends an “ACK\_LEAVE” message to $i$ (line 17). In Fig. 2.2 the message pattern and the consequent changing of connectivity relations is described.

After line 17 (executed at time $t$), we have: (i)$\forall j \in i\.children, j \in k\.children$, then $k \rightarrow t j$ and (ii) $k \not\rightarrow t i$ by eliminating $i$ from $k\.children$. When at time $t'$ the “ACK\_LEAVE” is received by $i$ (line 13), $i$ sets its neighbor variables to nil, then only the relation $k \rightarrow t' j$ holds. When at time $t''$ “NEW\_PARENT” ($k$) is received by each $j$ (lines 18-19), then the following relations hold: $k \rightarrow t'' j \land j \rightarrow t'' i$.

In a more difficult case, the $i$’s parent $k$ is leaving when the “LEAVE” message is received. In that case an order on the departures of $i$ and its parent $k$ is needed, to break the tie. The rule is the following: if $i$ and its parent $k$ have concurrent leaves, $k$ leaves the system before $j$. In this case, while a step is running for $i$, a basic step succeeds for $k$. Then a new (strong) connectivity relation involves $i$ and the $k$’s parent $j$, i.e. $i \rightarrow t'' j \land j \rightarrow t'' i$.

A this point $i$ will have to start a new step since the variable $parent\_change$ has been set to true (line 19).

In the worst case scenario all ancestors of $i$ are leaving processes. In this case the tree root is also involved in a leave, and thus a new root has to be found.

This can be done by considering an execution of a level-order traversal algorithm launched by the root whose aim is to return the first process whose state is in (if any). By the fact that leaving processes does not add in their neighbor variables new children neither coming from the outside (line 14) nor already in the tree (line 16) and by finite (even though unbounded) concurrency, the level-order traversal algorithm execution eventually terminates. It is worth to note that if all processes in the tree are simultaneously leaving, the level-order traversal algorithm terminates but a live-lock arises since a new root has

---

Figure 2.2: Change of topology and message pattern for leaving
not been found [34]. In this case different strategies may be pursued, e.g. the root could unblock all leaving processes in an arbitrary order and could switch to in waiting for a new child before trying to leave again.

For sake of simplicity we consider only the case where the root is a stable process avoiding to consider the level-order traversal algorithm. In this case the number of steps, before i ends successfully a basic step (at line 13), is bounded by its depth in the tree.

```plaintext
connectivityManagement;
1 var : timeout := 0 Timeout;
2 parent_change := ⊤ Boolean;

3 when ((s = joining) and (timeout = 0)) do
4 a := contact();
5 send ["JOIN"] to a;
6 set timeout = T;
7 when (receive ["ACK_JOIN"] from a) do
8 parent := a; s = in;

9 when ((s = leaving) and (parent_change)) do
10 send ["LEAVE"(children)] to parent;
11 parent_change := ⊥;
12 when (receive ["ACK_LEAVE"] from parent) do
13 parent := nil; ∀k ∈ children := nil; s := out;

14 when ((receive ["JOIN"] from j) and (s = in)) do
15 children := children ∪ {j}; send ["ACK_JOIN"] to j
16 when ((receive ["LEAVE"(j.children)] from j ∈ children) ∧ (s = in)) do
17 children := children ∆ j.children − {j}; send ["NEW_PARENT"(i)] to each j.children; send ["ACK_LEAVE"] to j;
18 when (receive ["NEW_PARENT"(j)] from a) do
19 parent := j; parent_change := ⊤;
```

Figure 2.3: The Basic Protocol at i

The False Children Problem.

It is important to remark that, due to asynchrony, the set children could include some false children: let us suppose that upon the i's join, j is the first contact retrieved by the contact() function. Then, j sends an “ACK_JOIN” to i, but the timeout on i elapses. Then, k is the second contact retrieved by the contact(), k sends an “ACK_JOIN” to i before the timeout elapses. In this case, i.parent = k, but i is recorded in k.children and in j.children (actually, i is a “false” child of j). Fig. 2.4 shows the message pattern and corresponding overlay changes.

---

3If the root left after the unblocking of the leaving processes the tree would become empty falling in Impossibility 1.3.8.
The presence of false children creates an asymmetry between the directed tree $T(t)$ defined by \textit{parent} variables and the digraph (which contains the tree $T(t)$) defined by the \textit{children} variables. Fig. 2.5 shows a possible overlay containing false children pointing out two different graphs, the graph defined by \textit{parent} variables and the graph defined by \textit{children} variables.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_5}
\caption{Asymmetry in the overlay defined by parent and children variables}
\end{figure}

To make the two graphs isomorphic, if the system were static all that would be needed for each child is to respond with either a \textit{parent} or \textit{non-parent} message. In this case, the graph is no more a tree because of the presence of two paths connecting a pair of vertices. The tree construction has some similarities with the Spanning Tree Construction for static systems [36].

\footnote{The graph is no more a tree because of the presence of two paths connecting a pair of vertices.}
\footnote{The tree construction has some similarities with the Spanning Tree Construction for static systems [36].}
however, both children and parents are supposed to be transient, i.e. the may leave the tree at any arbitrary point of time. This makes hard to remove false children.

This basic protocol assumes that no false child is ever created in any run. Then it is presented a protocol variant that works with false children but in which false children are not removed. Then, techniques to remove false children will be analyzed.

2.1.2 Proof of the Protocol

Let us remark that for the sake of simplicity, the proof of the protocol is based on the following assumptions:

Assumption 2.1.1. The initiator of the overlay r is a stable vertex.

Assumption 2.1.2. No false child is ever created.

Lemma 2.1.3. At any time \( t \), the Basic Protocol arranges each vertex \( i \in O(t) \) in a connected digraph \( T(t) \) defined by parent and children variables.

Proof. The proof is by induction on the connected digraph \( T(t) \).

Base case. As long as there are no state transitions, \( T(t_0) = \{(r, \{r.parent = r, r.children = \text{nil}\})\} \).

Induction. Suppose that all vertices in \( O(t) \) belong to the connected digraph \( T(t) \) defined by their parent and children variables.

The proof first considers a prefix of any run in which the only transitions of the variable \( s \) are \([\text{out/joining}]\) transitions. Then, any prefix is extended to comprise even the \([\text{in/leaving}]\) transitions of \( s \).

\([\text{out/joining}]\) transitions of \( s \). Let us suppose that \( s_i \) changes from \text{out} to \text{joining} at time \( t \) at a process \( i \).

When \( i.parent \) changes its state from \text{nil} to \( j \), the Basic Protocol had already established (in the following order): (i) one weak connection between \( i \) and a vertex \( j \) such that \( i \in j.children \), (ii) a strong connection by establishing the weak connection \( i.parent = j \).

Then, when at time \( t \), \( i \) becomes a vertex of \( O(t) \), \( i \in T(t) \).

\([\text{in/leaving}]\) transitions of \( s \). Let us suppose that \( s_i \) changes from \text{in} to \text{leaving} at time \( t \) at a process \( i \).

Let us suppose that at time \( t \) (i) \( i \) belongs to a branch \( B(t) \) of \( T(t) \) (ii) the branch has depth \( d \) and (iii) \( i \) is at level \( \ell > 0 \) with with \( d, \ell \) finite and unknown (by finite and unbounded concurrency).

Let \( B(t) = \{k_\ell, \ldots k_1, k_m, k_{m-1}, \ldots k_0\} \) such that \( k_0 = r, i = k_\ell, k_{m-1} = k_m.parent (m = 1 \ldots d - 1) \).

**case1.** Suppose that \( i \) is the only vertex which triggers a transition \([\text{in/leaving}]\) in \( B(t) \).
The deletion of \( i \) from the overlay would partition \( B(t) \) in two disconnected sub-branches \( B(t)' = \{k_d, \ldots k_{t-1}\} \), \( B(t)'' = \{k_{t+1} \ldots k_0\} \).

The Basic Protocol assures that when \( i \) leaves, at least the weak connection \( k_{t-1} \to k_{t+1} : k_{t+1} \in k_{t-1}.children \) is established. In particular, \( i \) sends the message “LEAVE(\( i \).children)” to \( k_{t-1} \) and remains in the overlay until an answer is received. When the “LEAVE” message is received, since \( k_{t-1}.s = in \), \( k_{t-1} \) inserts \( k_{t+1} \) in \( k_{t-1}.children \) and an “ACK_LEAVE” message is sent from \( k_{t-1} \) to \( i \). Thus, when \( i \) is deleted from the overlay, \( k_{t+1} \) and \( k_{t-1} \) are still (weak) connected, i.e. \( k_{t-1} \) and \( k_{t+1} \in T(t) \).

**Case 2.** Suppose that all processes \( k_m \) with \( \ell \leq m \leq 1 \) concurrently trigger a transition [in/leaving] in \( B(t) \).

The simultaneous deletion of these vertexes \( k_m \) from the overlay would partition \( B(t) \) in two disconnected sub-branches \( B(t)' = \{k_d \ldots k_{t-1}\} \), \( B(t)'' = \{k_0\} \).

The Basic Protocol provides that each process \( k_m \) with \( \ell \leq m \leq 1 \) sends a “LEAVE \( (k_m,.children) \)” message to \( k_{m-1} \) (line 10).

All vertices \( k_{m-1} \) with \( m > 1 \) discard any “LEAVE\( (k_m,.children) \)” while they have \( k_{m-1}.s = \text{leaving} \). Thus, each \( k_m \) remains in the overlay at least until \( k_1 \) receives an “ACK_LEAVE” message (line 12) sent by \( k_0 \) (line 17) \((k_0.s = in \) by Assumption 2.1.1). As in case 1, when \( k_1 \) receives this acknowledgment a weak connection between \( k_2 \) and \( k_0 \) : \( k_2 \in k_0,.children \) has been established. Thus, each \( k_m \) with \( \ell \leq m \leq 1 \) belong to a weak connected branch \( B(t_1) = \{k_d, \ldots k_\ell, \ldots k_m, k_{m-1}, \ldots k_2, k_0\} \) with \( t_1 \) the time in which \( k_1 \) leaves.

At line 17, the vertex \( k_0 \) sends also a “NEW_PARENT” message to \( k_2 \). Note that, the “LEAVE\( (k_2,.children) \)” message cannot be acknowledged until the “NEW_PARENT” message from \( k_0 \) is received. Only when this message is received \( k_2 \) sets \( k_2,.parent \) to a vertex with the variable \( s = in \). At this point \( k_2 \) retransmits “LEAVE\( (k_2,.children) \)” to the new parent \( k_0 \). From the time in which \( k_0 \) receives the “LEAVE\( (k_1,.children) \)” message to the time in which \( k_2 \) receives the “ACK_LEAVE” message (from \( k_0 \)), each \( k_m \) with \( \ell \leq m \leq 1 \) belong to a weak connected branch \( B_1 = B(t_1) \).

When the “ACK_LEAVE” message arrives to \( k_2 \) at \( t_2 \), each \( k_m \) with \( \ell \leq m \leq 1 \) belong to a weak connected branch \( B(t_2) \) s.t. \( k_\ell \in k_0,.children \), and so on until \( k_{t+1} \) becomes a child of \( k_0 \) after the leaves of all processes \( k_m \) with \( \ell \leq m \leq 1 \). Then all \( k_m \) processes with \( \ell < m \leq d \) are weak connected and belong to \( T(t') \) for each \( t < t' < t'' \) where \( t'' \) is the time in which \( k_t \) leaves.

**Case 3.** Suppose that not all processes \( k_m \) with \( \ell \leq m \leq 1 \) concurrently trigger a transition [in/leaving] in \( B(t) \), but only a number \( l < m \). In this case the branch \( B(t) \) can be subdivided in sub-chains in which the extremes are processes not leaving. Then this case can be reduced to the previous ones.

**Theorem 2.1.4 (Active Weak Connectivity).** At any time \( t \),

\[
\forall i, j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t) \Rightarrow (i \xrightarrow{t} j) \lor (j \xrightarrow{t} i)
\]

**Proof.** By Lemma 2.1.3, the Theorem follows.
Theorem 2.1.5 (Global Progress). If \( \exists i \in \Pi \) that executes the join action at time \( t \), then \( \exists t', j : t' > t \land \langle j, j.X \rangle \in O(t') \land \langle j, j.X \rangle \not\in O(t) \)

Proof. If a process executes the join, it sets \( s = \text{joining} \). The Basic Protocol tries to establish a connection with a vertex by getting \( a \) from contact(). Let us suppose, by contradiction, that no other process \( j \neq r \) at time \( t' > t \) belongs to the overlay. By Assumption 2.1.1, \( r \) is a vertex at any time. If no other process \( j \) is in the overlay, since the contact() function returns vertices, \( a \) is the root \( r \). By channel reliability and Assumption 2.1.1, eventually \( i \) sets \( i.parent = r \) and \( i.s = \text{in} \). Contradiction.

With the following Assumption even No-Lockout property is guaranteed:

Assumption 2.1.6. The contact() function eventually returns a stable vertex.

Theorem 2.1.7 (No-Lockout). If \( \exists i \in \Pi \) that executes the join action at time \( t \), then \( \exists t' : \langle i, i.X \rangle \in O(t') \)

Proof. If a process executes the join, it sets \( s = \text{joining} \). The Basic Protocol tries to establish a connection with a vertex by getting \( a \) from contact(). Let us suppose, by contradiction, that there exists no time \( t' > t \) such that \( i \) belongs to the overlay. By Assumption 2.1.6, eventually \( a \) is a stable vertex. By channel reliability eventually \( i \) sets \( i.parent = a \) and \( i.s = \text{in} \). Contradiction.

Corollary 2.1.8. The Basic Protocol assures that the overlay never partition.

The Basic Protocol and Strong Connectivity. It is worth to note that the Basic Protocol assure Weak Connectivity at any point of time and Strong Connectivity most of the time. Actually, the Basic Protocol builds a strong connected graph that is lost during the updating of the overlay due to the leave of a vertex. In any run in which leaves eventually subside, the Basic Protocol assures Eventual Strong Connectivity.

Corollary 2.1.9. If joins eventually subside, The Basic Protocol assures eventual strong connectivity.

Proof. (Sketch) If joins eventually subside, overlay changes eventually subside because the finite number of vertices that can leave. After all leaves, all vertices are stable and are weak connected thanks to Lemma 2.1.3. Now we prove that any pair of vertices is also strong connected.

By Contradiction, let us suppose that there exists a run \( R \) in which \( i \) joins and leaves subside at time \( t \) and a pair \( i,j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t) \) and \( i \rightarrow_t j \) and \( i \neq_t j \). This implies that in the path from \( i \) to \( j \) there exists at least one pair \( k,h \) such that: \( s_k = \text{in}, s_h = \text{in}, k \in h.X \) and \( h \not\in k.X \). Without loss of generality suppose that \( k = h.parent \) but \( h \not\in k.children \). Now consider a run \( R' \) with prefix equal to \( R \) and with only one more [leaving/out] transition triggered by \( k \) at time \( t' > t \). By the protocol if \( h \not\in k.children \) then \( h \) remains disconnected from the tree after the leave of \( h \). By Lemma 2.1.3 no run \( R' \) can occur, then an absurd arises.
2.1.3 Protocol Variants Handling False Children

We will present four variants of the Basic Protocol handling false children.

A Protocol Tolerating False Children.

The following protocol, called protocol A (Fig. 2.6), is obtained modifying lines 16-19 of the basic protocol in Fig. 2.3. The presence of false children is tolerated: false children are not removed and the protocol deals with their presence. In particular, by piggybacking the identifier of the parent which is leaving in the “NEW_PARENT” message, the children is able to understand if it is a real child of the leaving parent or not (line 16). Only in the former case it will update the parent variable.

```
15 ... when ((receive ["LEAVE"(j.children)] from j ∈ children) ∧ (s = in)) do
16 ... send ["NEW_PARENT"(i, j)] to j.children;...
17 when (receive ["NEW_PARENT"(new, old)] from a ∧ i.parent=old) do
18   parent := j; parent_change := T;
```

Figure 2.6: Protocol A at i

It is worth to note that without removing false children, a stable process may have to store an infinite number of neighbors.

Techniques to Remove False Children

The importance of removing false children lies in the possibility of maintaining a tree invariant both for parent and for children variables. This could be useful, for instance, to obtain (during the stability periods of the overlay in terms of joins/leaves) the same complexity for (i) the broadcast of a message from r to all other vertices in the overlay, and for (ii) the convergecast used, for instance, to allow r to get an acknowledgment sent by any leaf.

The convergecast, exploiting the tree defined by parent variables, requires (i) a number of messages \(O(n)\) where \(n\) is the (unbounded but finite) number of vertices in the overlay, and (ii) a time complexity \(O(h)\) where \(h\) is the (unbounded and finite) height of the tree [36].

If children variables do not define a (same) tree the presence of false children may heavily impact the broadcast complexity. In the worst case scenario, any vertex \(i\) is a false child of all vertices which belonged to the overlay when \(i\) joined.

In Fig. 2.7 a scenario in which the overlay grows until six vertices is shown.

In this scenario, each process accesses the overlay sequentially in the order 1,2...6. Each process \(i\) becomes false child of all vertices \(k\) with \(1 ≤ k < i\) due to asynchrony. Each \(i\) process has a number of outgoing arcs equal to \(n - i\). This implies that a broadcast requires a number of messages \(O(n^2)\) \(^6\). Time complexity is still \(O(h)\).

---

\(^6\)This upper bound is computing considering that messages arrived from non – parents are not further diffused.
Protocol B. The protocol B (Fig. 2.10) is obtained modifying the $[\text{joining/in}]$-transition of the basic protocol in Fig. 2.3. The basic idea is to send a “NON-PARENT” message to all processes that have been contacted but have not been selected as parents. Unfortunately, this simple idea maybe very dangerous in a dynamic system. Fig. 2.8 describes a scenario in which a process $k$ receives a “NON-PARENT” message because it had a false children $i$. In the meanwhile, however, it has become the new parent of $i$, because one of $k$’s children, namely $j$ (i) was the parent of $i$ and (ii) $j$ has left the overlay.

The use of sequence numbers (lines 3, 7, 11) avoids that the delayed “NON-PARENT” message wrongly deletes the new child $i$ of $j$. Fig. 2.9 shows the same message pattern of Fig. 2.8 with the use of sequence numbers provided by the protocol.

In the protocol $B$, the removal of a false child $i$ is assured only if $i$ has contacted during the $[\text{joining/in}]$ transition only stable vertices (or a vertex that remain in the tree for a time long enough to be updated). If, however, $j$ has a false child $i$ and $j$ leaves before receiving the “NON-PARENT” message, $i$ is inherited as a false child by another vertex. In this case $i$ will never be removed.

Protocol C: Eventual Tree. Protocol C (Fig. 2.11) is an improved version of protocol B. Protocol C is able to remove also inherited (but stable) false children of stable vertices. This is possible, exploiting “NEW_PARENT(new,old)” messages (lines 13, 14). When $i$ receives such a message in which old is a non-parent vertex previously contacted, it can retransmit the “NON_PARENT” message to new, i.e. the vertex that has inherited $i$ as false child.

Protocol $C$ assures that all stable vertices eventually remove (inherited) false children that are stable vertices.
Lemma 2.1.10. Protocol C assures that all stable vertices eventually remove all stable false children.

Proof. By Contradiction. Let us suppose that there exists a process i such that: i is a stable vertex, i inserts a false child j at time t, j belongs to i.children in the time interval [t, ∞), j is a stable vertex. This implies that i never receives a “NON_PARENT” message from j. Two cases may occur:

Case a. i inserts j in i.children upon receiving a “JOIN” message from j. In this case, after j has set its parent ≠ i (by assuming that j is a false child), it has already sent a a “NON_PARENT” to i. By channel reliability an absurd arises.

Case b. i inserts j in i.children upon receiving a “LEAVE(j)” message from k. In this case, i sends a “NEW_PARENT(i, k)” to j. Suppose that k does not belong to j.contacted. It means that j (i) has never send “JOIN” message to k and (ii) j has never received a “NEW_PARENT(k, h)” with h ∈ contacted. Iterating this reasoning backwards to infinity we get that j has contacted = ∅. In this case, j is not a false child of anyone contradicting the initial hypothesis.

Theorem 2.1.11. If joins eventually subside, protocol C assures that all vertices in the overlay eventually form an undirected tree defined by parent and children variables.
Figure 2.9: Safe receiving of leave and non-parent messages thanks to sequence numbers

Proof. If joins eventually subside, overlay changes eventually subside because the finite number of vertices that can leave. After all leaves, all vertices are stable and are strongly connected thanks to Corollary 2.1.9. At this point false children still in the overlay are stable and are stored only in stable vertices. Then the Theorem follows directly from Lemma 2.1.10.

Protocol D: Removal of transient false children. Protocol C is able to remove only stable children. Any transient false child, however, may leave before sending a “non-parent” message. Then, any transient false child may be stored forever in stable vertices, i.e. any stable vertex $i$ may have an infinite set of dangling arcs $(i, \text{nil})$ for each transient child which has left. This implies that the required memory for stable vertices is infinite.

The following protocol (Fig. 2.12) embeds a technique allowing all stable vertices to eventually remove all transient false children. The technique relies on the flooding of messages informing what vertices had left the system. When a process receives such a message it can check if one of its (false) children had left. If so, the false child can be deleted. The flooding is realized through two primitives: Broadcast and Convergecast (Fig. 2.13, 2.14, 2.15, 2.16). Broadcast is a flooding initiated by the root, while Convergecast is a diffusion from a vertex in an arbitrary position of the tree (directed tree defined by parent variables) to the root that occurs only over the path connecting the vertex to the root. Note that during an overlay change, the diffusion of a message could be interrupted. In fact only in vertices relay messages (Fig. 2.14, Fig2.16, line 1). If a message $m$ arrives to
connectivityManagement_i:
1 var : timeout := 0 Timeout;
2 parent_change := ⊤ Boolean;
3 seq := 0 Integer;
4 previous := nil Neighbor;

5 when ((s = joining) and (timeout = 0)) do
6 a := contact();
7 seq := seq + 1; send ["JOIN"(seq)] to a;
8 send ["NON-PARENT"(seq)] to previous; previous := a set timeout = T;

9 when (receive ["JOIN"] from j) and (s = in)) do
10 children := children ∪ (j, seq); send ["ACK_JOIN"] to j
11 when (receive ["NON-PARENT"(seq)] from j ∈ children) do
12 children := children − {j};
13 ...

Figure 2.10: Protocol B at i

connectivityManagement_i:
1 var : timeout := 0 Timeout;
2 parent_change := ⊤ Boolean;
3 seq := 0 Integer;
4 previous := nil Neighbor;
5 contacted := ∅ set of type Π;

6 when ((s = joining) and (timeout = 0)) do
7 a := contact();
8 seq := seq + 1; send ["JOIN"(seq)] to a;
9 send ["NON-PARENT"(seq)] to previous; previous := a set timeout = T;

10 when (receive ["JOIN"] from j) and (s = in)) do
11 children := children ∪ (j, seq); send ["ACK_JOIN"] to j
12 when (receive ["NEW_PARENT"(new, old)] from a ∨ ∃j ∈ contacted : j = old) do
13 contacted := contacted ∪ {new};
14 send ["NON-PARENT"(seq : seq ∈ (j, seq))] to new
15 when (receive ["NEW_PARENT"(seq)] from j ∈ children) do
16 children := children − {j};
17 ...

Figure 2.11: Protocol C at i
a leaving process $i$ (this can happen when the “LEAVE” message of $i$ is still in transit at the moment in which the $i$’s parent relays $m$ to its children) it is no further diffused by $i$ to its sub-tree. However, both primitives react to overlay changes: through $parent\_change2$ and $children\_change$ every in vertex can detect when neighbors have changed. When this happens, the vertex retransmits the message $m$ to new neighbors (Fig. 2.15, line 4 and Fig. 2.13 line 4).

```plaintext
connectivityManagement;
1 var : timeout := 0 Timeout;
2 parent\_change := ⊤ Boolean;
3 parent\_change2 := ⊥ Boolean;
4 children\_change := ⊥ Boolean;

5 when ((s = joining) and (timeout = 0)) do
6 a := contact();
7 send ["JOIN"] to a;
8 set timeout = T;
9 when (receive ["ACK\_JOIN"] from a) do
10 parent := a; s = in;

11 when ((s = leaving) and (parent\_change)) do
12 send ["LEAVE"(children)] to parent;
13 parent\_change := ⊥;
14 when (receive ["ACK\_LEAVE"] from parent) do
15 parent := nil; ∀k ∈ children := nil; s := out;

16 when ((receive ["JOIN"] from j) and (s = in)) do
17 children := children ∪ {j}; send ["ACK\_JOIN"] to j;
18 when ((receive ["LEAVE"(j,children)] from j ∈ children) ∧ (s = in)) do
19 children := children ∪ j.children − {j}; send ["NEW\_PARENT"(i,j)] to j.children;
20 children\_change := ⊤;
21 send ["ACK\_LEAVE"] to j;
22 if (i.father = father) then Broadcast("REMOVE(j)");
23 else Convergecast("REMOVE(j)")
25 when (receive ["NEW\_PARENT"(new,old)] from a ∧ i.parent = old) do
26 parent := j; parent\_change := ⊤; parent\_change2 := ⊤;
28 children := children − {j};
29 when ((CReceive(m_e)) ∧ (i.father = father)) do
30 Broadcast(m_e);
```

Figure 2.12: Protocol D at $i$

The flooding assures that each message sent at time $t$ will eventually arrive to all stable vertices $i : i ∈ O(t)$.

**Lemma 2.1.12.** Each message sent at time $t$ through a Broadcast or Convergecast primitive eventually arrive to all stable vertices $i : \langle i, i.X \rangle ∈ O(t)$.
Broadcast($m_b$)
1  var : $tx_b := \text{Queue};$

2  send [$m_b$] to children;
3  add($m_b$) to $tx_b$;
4  when (children_change = $\top$) do
5    $\forall m_b \in tx_b$ send [$m_b$] to children;
6    children_change = $\bot$;

Figure 2.13: The Broadcast Protocol at $i$

BReceive($m_b$)
1  when ((receive [$m_b$] from a) $\land$ (a = father) $\land$ (s = in)) do
2    if (children $\neq$ nil) then Broadcast($m_b$);

Figure 2.14: The BReceive Protocol at $i$

Convergecast($m_c$)
1  var : $tx_c := \text{Queue};$

2  send [$m_c$] to father;
3  add($m_c$) to $tx_c$;
4  when (parent_change2 = $\top$) do
5    $\forall m_c \in tx_c$ send [$m_c$] to parent; parent_change2 = $\bot$;

Figure 2.15: The Convergecast Protocol at $i$
\texttt{CReceive}(m_c)

\begin{verbatim}
1   when ((receive \text{[m_c]} from a) \land (s = in)) do
2       if (father \neq i.father) then Convergecast(m_c);
\end{verbatim}

Figure 2.16: The CReceive Protocol at \(i\)

\textit{Proof.} By Contradiction. Let us suppose that (i) a message \(m\) is flooded starting from \(i\) at time \(t\), (ii) there exists a stable vertex \(j : \langle j, j.X \rangle \in O(t)\) and \(j\) never receives \(m\). If \(j\) never receives \(m\), no neighbor of \(m\) ever sends \(m\) to \(j\). It means that (i) \(m\) is never received by any neighbor of \(j\) or (ii) all neighbors of \(j\) are leaving/out when the message arrives (from line 1 of Breceive and Creceive follows that only \textit{in} processes relay the message). By iterating the former case until \(i\) is reached, we have that all neighbors of \(i\) do not relay \(m\) to their neighbors, i.e. all neighbors of \(i\) are leaving/out when the message arrives. In this case, however, \(i\) will receive a "\texttt{LEAVE}" for each children and a "\texttt{NEW\_PARENT}" message from its parent. Two cases may occur: (1) \(i\) is the root, (2) \(i\) is not the root. In case (1) \(m\) was sent trough a \texttt{Broadcast} (line 23), in case (2) \(m\) was sent with a \texttt{Convergecast} (line 24). When \(i\) receives a "\texttt{LEAVE}" message from a child, then \texttt{children\_change} is set to \texttt{true} and then a new broadcast of all messages in \(tx_b\) initiates (lines: 23, 29-30). When \(i\) receives a "\texttt{NEW\_PARENT}" message from its parent, then \(i\) sets \texttt{parent\_change2} = \texttt{T} and a convergecast towards the root of all messages in \(tx_c\) initiates. In both cases \(m\) is already in \(tx_b\) or \(tx_c\) respectively, then a flooding towards new neighbors of \(m\) restarts. By iterating this case, eventually \(j\) is a new neighbor of \(i\) with \(s = \text{in}\). Then eventually \(j\) will receive \(m\). Contradiction.

Assuming that each "\texttt{JOIN}" message, sent by \(i\), is received before the message \(m\) informing about the \(i\)'s leave, all \(j\) processes having \(i\) as false child are already in the overlay when the diffusion starts. This assures that \textit{all stable vertices can eventually remove all false (transient) children}. Formally,

\textit{Assumption 2.1.13.} \footnote{An implementation of this assumption can be as follows: A process \(j\) stores in a set \texttt{removed} each processes \(j\) from which \(j\) receives a "\texttt{REMOVE}(i)". \(j\) then discards all JOIN messages received from processes in \texttt{removed} and removes \(i\) from \texttt{removed}. To maintain finite memory the "\texttt{REMOVE}(i)" message can piggyback the \(i\)'s \texttt{contacted}. In that case, upon receiving "\texttt{REMOVE}(i, i.\texttt{contacted})", \(j\) stores \(i\) into \texttt{removed} only if \(j\) is in \(i.\texttt{contacted}\).} If a process \(j\) receives "\texttt{REMOVE}(i)" message and \(i\) has previously sent a "\texttt{JOIN}" message to \(j\), then the "\texttt{JOIN}" message arrives before the "\texttt{REMOVE}(i)" message.

\textit{Lemma 2.1.14.} Protocol \(D\) assures that \textit{all stable vertices eventually remove all false transient children.}

\textit{Proof.} By Lemma 2.1.12 any "\texttt{REMOVE}(i)" sent at time \(t\), will arrive to all stable vertices belonging to the overlay at time \(O(t)\). Thus suppose by contradiction that there exists a stable vertex \(j\) such that \(i \in j.\text{children}\) and \(i.\text{parent} \neq j\) in \(O(t')\) with \(t' > t\). By Assumption 2.1.13 all stable processes \(k\) present at time \(t\) are \(k \neq j\). Contradiction. \qed
Broadcast/Convergecast Implementation. The presented Broadcast/Convergecast implementation relies on an infinite memory. By assuming FIFO channels it is possible avoiding the retransmissions of messages sent in Broadcast. When a parent receives a “LEAVE” message it has the new children to send messages to. All previous messages can be considered relayed by the leaving child if (i) the “ACK_LEAVE” is the last message sent to the leaving child and (ii) the leaving child can relay messages even when \( s := \text{leaving} \).

On the contrary, the Convergecast implementation cannot avoid retransmissions, as during the messages exchange between a leaving vertex and its parent, children of the leaving vertex may stay without a parent for a while (between the “ACK_LEAVE” arrival at the leaving vertex and the arrival of the “NEW_PARENT” message to the children of the leaving vertex). In this case messages sent during this period may be lost.

The AW problem and Infinite Memory. The Basic Protocol assumes that no false child is ever created. However, the more the asynchrony of a system, the more the probability of having false children. Tolerating false children means relying on infinite memory. Removing false children requires infinite memory for the broadcast/convergecast implementation.

The only way to avoid false children (and then the use of infinite memory) is by implementing a [joining\([/\text{in}]]\] transition in which only one contact is selected and an answer from this contact is waited. No-lockout in this case is guaranteed strengthening Assumption 2.1.6 by assuming that the contact function always returns stable vertices.

2.1.4 Maintenance of a Topology

The work of Liu, Misra, Plaxton [34] presents a maintenance protocol in charge of maintaining a ring topology in absence of failures. They consider an asynchronous system with unbounded concurrency and a finite arrival model [38]. They rely on a contact() function similar to the one presented in Section 2.1. They show that the the ring invariant can be assured only in stabilization periods of the overlay. Stabilization is intended with respect the arrival pattern: if joins eventually subside then the overlay eventually converges to a ring.

In the above Section we have reached similar results for the maintenance of a tree. Nevertheless, in [34] the problem of the undesired connections established upon a join due to asynchrony does not arise. This is due to some shortcomings restricting asynchrony. In particular, they first assume that even processes no longer in the overlay responds to protocol messages. This allows the design of a join protocol in which a joining process sends only one JOIN message. The termination of the join protocol (No-Lockout) is assured since the JOIN message is eventually acknowledged or declined.

Then, they weaken this assumptions and change the leave protocol accordingly. The join protocol, however, is not modified. It turns out in assuming any joining process able to select contacts that will not be in an out state while the JOIN message is in transit. This shortcoming allows a joining process to ever get an answer to its join request by selecting only one contact. In this way, the creation of not desired edges, like false children, is avoided.
2.2 A Protocol Implementing Active Strong Connectivity

Now, we consider how to implement AS. In the following the overlay will be denoted as $K(t)$. Each element of $K(t)$ is composed by a pair $\langle i, (i.parent, i.oldparents, i.children) \rangle$ where each $j \in i.parents$ and each $k \in i.children$ are neighbor variables. The AS protocol (Fig. 2.23) executes a different update with respect the Basic Protocol: when a process is informed about new parent/children it simply adds these new neighbors without removing anything. Moreover, a leaving process $j$ sends a “LEAVE” message to both parent and children, then waits an acknowledgement from them. Note that the above protocol may trivially maintains active strong connectivity relying on infinite memory to store neighbor variables. To avoid of using infinite memory for neighbor variables, removal of neighbors that have left the overlay is required. The actual removal of a leaving vertex $j$ relies on the diffusion of “REMOVE($j$)” messages. The diffusion of a “REMOVE($j$)” message starts with a proper message, namely “START_REMOVE($j$)”, sent by the parent of $j$, namely $k$, to the new children. Logically, the “START_REMOVE($j$)” has to arrive after the “LEAVE($k$)” at any child $i$ (to avoid to remove a parent). To assure that, upon receiving “START_REMOVE($j$)”, $i$ waits 8 to have $j$ in oldparents (lines 28,29), i.e., the “REMOVE($j$)” starts after the “LEAVE($k$)” from $j$ is arrived. Fig. 2.17 shows the message pattern and corresponding overlay changes.

Flooding Implementation. Note that the information about the removing of $i$ is flooded. It allows to remove $i$ from all children variables that contains $i$ as a false child. Note that removing false children is even necessary to avoid that a leaving process waits forever an acknowledgement from a false child which has already left the overlay. The connectivity assured by the protocol allows to implement both Broadcast and Convergecast assuming FIFO (as discussed for the Broadcast in case of Protocol D.) The implementation is shown in Figures 2.18,2.19,2.20,2.21.

Handling Concurrency. Concurrency of leaves on the same branch has to be carefully handled. The mechanism implemented by the AS protocol with two different leave messages sent to both parent and children creates many races when two (or more) vertices leave concurrently. Fig.2.22 shows two different scenarios. In Fig. 2.22(a) $j$ sends two leave messages to its parent and its child, namely “LEAVE($i$)” to its parent and “LEAVE($k$)” to its child. The child updates its neighbor variable parent to $k$ by receiving the “LEAVE($k$)” message. Then $i$ decides to leave, and it sends a message “LEAVE($h$)” to its new parent $k$. The race between “LEAVE($i$)” and “LEAVE($h$)” could bring “LEAVE($h$)” to arrive first. In this case $k$ receives a “LEAVE($h$)” from a vertex which is not recognized as one of its children. In Fig. 2.22(b) is shown a symmetric scenario in which the $j$’s parent $k$ updates its neighbor variable children including $i$ by receiving the “LEAVE($i$)” message. Then $k$ decides to leave, and it sends a message “LEAVE($h$)” even to its new child $i$. The race between “LEAVE($k$)” and “LEAVE($h$)” could bring “LEAVE($h$)” to arrive first. In this case $i$ receives a “LEAVE($h$)” from a vertex which is not recognized as its parent. The

8 Upon receiving any different message a thread is spawned.
The protocol handles these races by ordering leave messages (lines 23, 28).

\[ \text{LEAVE}(i) \]
\[ \text{ACK\_LEAVE} \]
\[ \text{START\_REMOVE}(j) \]
\[ \text{oldparents}\{-i\} \]
\[ \text{parent}=K \]

**Figure 2.17:** Changes of topology and message pattern for leaving

\[ \text{Broadcast}(m_b) \]
1. send \([m_b]\) to children;

**Figure 2.18:** The Broadcast Protocol at \(i\)

\[ \text{BReceive}(m_b) \]
1. when \(((\text{receive } [m_b] \text{ from } a) \land (a = \text{father}))\) do
2. if \((\text{children} \neq \text{nil})\) then \(\text{Broadcast}(m_b)\);

**Figure 2.19:** The BReceive Protocol at \(i\)
Convergecast($m_c$)
1 send $[m_c]$ to $parent \cup oddparents$;

Figure 2.20: The Convergecast Protocol at $i$

CReceive($m_c$)
1 when (receive $[m_c]$ from a) do
2 if (parent $\neq i$.parent) then Convergecast($m_c$);

Figure 2.21: The CReceive Protocol at $i$

Figure 2.22: Races among leave messages
connectivityManagement
1 var timeout := 0 Timeout;
2 parent_change := ⊤ Boolean;
3
4 when ((s = joining) and (timeout = 0)) do
5 a := contact();
6 send ["JOIN"] to a;
7 set timeout = T;
8 when (receive ["ACK.Join"] from a) do
9 parent := parent ∪ a; s = in;
10 when ((s = leaving) and (parent_change)) do
11 send ["LEAVE"(children)] to parent; send ["LEAVE"(parent)] to children;
12 parent_change := ⊥;
13 when ((receive ["ACK.LEAVE"] from parent) ∧ receive ["ACK.LEAVE"] from k∀k ∈ children) do
14 parent := nil; ∀k ∈ children := nil; s := out;
15 when ((receive ["JOIN"] from j) and (s = in)) do
16 children := children ∪ {j}; send ["ACKJOIN"] to j;
17 when ((receive ["LEAVE"(j,children)] from j) ∧ (s = in)) do
18 wait until (j ∈ children);
19 children := children − j.children;
20 send ["ACK.LEAVE"] to j;
21 send ["START.REMOVE"(j)] to children
22 when (receive ["LEAVE"(new)] from j) do
23 wait until (j = parent);
24 oldparents := oldparents ∪ {parent};
25 parent := new;
26 parent_change := ⊤; send ["ACK.LEAVE"] to j;
27 when (receive ["START.REMOVE"(old)] from j) do
28 wait until (old ∈ oldparents);
29 Convergecast("REMOVE(j)");
30 when (BReceive("REMOVE(j)")) do
31 if ((j ∈ children))
32 then children := children − {j};
33 if ((j ∈ oldparents))
34 then oldparents := oldparents − {j};
35 when ((CReceive(mc)) ∧ (i.father = father)) do
36 Broadcast(mc);

Figure 2.23: A Protocol implementing AS at i
2.2.1 Proof of the Protocol

The proof of the protocol is based on the following assumptions:

**Assumption 2.2.1.** \(O(t_0)\) contains only one element \(\langle r, r.X \rangle\): \(r\) is stable, \(r.parent = r\), \(r.children = NIL\), \(r.oldparents = NIL\), \(s := in\).

**Assumption 2.2.2.** Channels are FIFO.

**Lemma 2.2.3.** At any time \(t\), the AS Protocol arranges each vertex \(i: \langle i, i.X \rangle \in O(t)\) in a connected graph \(K(t)\) defined by parent, oldparents and children variables.

*Proof.* The proof is by induction on the connected graph \(K(t)\).

**Base case.** As long as there are no state transitions, \(K(t_0) = \{ \langle r, r.parent = r, r.children = NIL, r.oldparents = NIL \rangle \}\).

**Induction.** Suppose that all vertices in \(O(t)\) belong to the connected graph \(K(t)\) defined by their parent, children and oldparents variables.

The proof first considers a prefix of any run in which the only transitions of the variable \(s\) are [out/joining] transitions. Then, any prefix is extended to comprise even the [in/leaving] transitions of \(s\).

**[out/joining] transitions of \(s\).** Let us suppose that \(s_i\) changes from out to joining at time \(t\) at a process \(i\). When \(i.parent\) changes its state from nil to \(j\), the AS Protocol had already established (in the following order): (i) one weak connection between \(i\) and a vertex \(j\) such that \(i \in j.children\), (ii) a strong connection by establishing the weak connection \(i.parent = j\). Then, when at time \(t\), \(i\) becomes a vertex of \(O(t)\), \(i \in K(t)\).

**[in/leaving] transitions of \(s\).** Let us suppose that \(s_i\) changes from in to leaving at time \(t\) at a process \(i\). Let us suppose that at time \(t\) (i) \(i\) belongs to a branch \(B(t)\) of \(T(t)\) (ii) the branch has depth \(d\) and (iii) \(i\) is at level \(\ell > 0\) with with \(d, \ell\) finite and unknown (by finite and unbounded concurrency). Let \(B(t) = \{ k_d, \ldots , k_\ell, k_m, k_{m-1}, \ldots , k_0 \}\) such that \(k_0 = r\), \(i = k_\ell\), \(k_{m-1} = k_m.parent\) \((m = 1 \ldots d - 1)\).

**Case 1.** Suppose that \(i\) is the only vertex which triggers a transition [in/leaving] in \(B(t)\). The deletion of \(i\) from the overlay would partition \(B(t)\) in two disconnected or weak connected sub-branches \(B(t)' = \{ k_d, \ldots , k_\ell-1 \}\), \(B(t)'' = \{ k_{\ell+1} \ldots k_0 \}\). The AS Protocol assures that while \(i\) leaves, the strong connection \(k_{\ell-1} \leftrightarrow k_{\ell+1} \in k_{\ell-1}.children\) is being established and the strong connections \(k_{\ell-1} \leftrightarrow i\) and \(i \leftrightarrow k_{\ell+1}\) are not lost. In particular, \(i\) sends the messages “LEAVE(i.children)” to \(k_{\ell-1}\) and “LEAVE(i.parent)” to \(k_{\ell+1}\) (line 11) and remains in the overlay until an answer is received from both (lines 12,14)). When the “LEAVE” message is received, both \(k_{\ell-1}\) and \(k_{\ell+1}\) have \(s = in\), then \(k_{\ell-1}\) inserts \(k_{\ell+1}\) in \(k_{\ell-1}.children\) (line 19) and \(k_{\ell+1}\) inserts \(i\) in \(k_{\ell+1}.oldparents\) and sets parent = \(k_{\ell-1}\) (lines 24,25). Both \(k_{\ell-1}\) and \(k_{\ell+1}\) send an “ACK_LEAVE” message to \(i\) (lines 20, 26).

Now we analyzes how \(i\) is removed.
(step 1) $i$ is removed by $k_{t+1}$ when the “START\_REMOVE($i$)” arrives to $k_{t+1}$. By line 28, $k_{t+1}$ removes $i$ from oldparents only when (i)$k_{t+1}.parent = k_{t-1}$ and by the fact that “START\_REMOVE($i$)” is sent by $k$ after receiving “LEAVE($i.children$)” when (ii) $k_{t+1} \in k_{t-1}.children$.

(step 2) $i$ is removed by $k_{t-1}$ after the previous step because of the diffusion of the “REMOVE($i$)” is started from $k_{t+1}$ at the end of the previous step.

Then, during the establishment of a strong connection between $k_{t-1}$ and $k_{t+1}$, the two strong connections between, respectively, $k_{t-1} \text{ and } i$ and between $i$ and $k_{t+1}$ are not removed. When $i$ is deleted from the overlay at time $t$, $k_{t+1}$ and $k_{t-1}$ are still strong connected, i.e. $k_{t-1}$ and $k_{t+1} \in K(t)$.

**case2.** Suppose that all processes $k_m$ with $\ell \leq m \leq 1$ concurrently trigger a transition [in/leaving] in $B(t)$. The simultaneous deletion of these vertexes $k_m$ from the overlay would partition $B(t)$ in two disconnected or weak-connected sub-branches $B(t) = \{k_0 \ldots k_{t-1}\}$, $B(t)'' = \{k_0\}$. The AS Protocol provides that each process $k_m$ with $\ell \leq m \leq 1$ sends a “LEAVE ($k_m.children$)” message to $k_{m-1}$ and a “LEAVE ($k_m.parent$)” to $k_{m+1}$ (line 11). All vertices $k_{m-1}$ with $m > 1$ have $k_{m-1}.s = \text{leaving}$ then they discard any incoming “LEAVE($k_m.children$)”. This implies that each $k_m$ with $\ell \leq m < 1$ will never receive an acknowledgement to its “LEAVE ($k_m.children$)” message from $k_{m-1}$. Only $k_1$ will receive an acknowledgment “ACK\_LEAVE” message from $k_0$ ($k_0.s = \text{in}$ by Assumption 2.2.1). $k_2$ even if it is leaving, will send an “ACK\_LEAVE” message to $k_1$ upon receiving “LEAVE($k_0$)” from $k_1$ (lines 22-26) and sets parent_change = $\top$ then will restart a basic step.

At this point two concurrent steps occur:

(step 1) the sending/receiving (from $k_2$) of “LEAVE($k_0$)”/“ACK\_LEAVE” at $k_1$

(step 2) the sending/receiving (from $k_0$) of the “LEAVE($k_2$)”/“ACK\_LEAVE” at $k_1$.

During these steps:

(i) as in case 1 is always maintained a strong connection between vertices

(ii) a “LEAVE($k_3$)” message sent by $k_2$ may arrive to $k_0$ (the (second) basic step undertaken by $k_2$ is concurrent with the basic step undertaken by $k_1$). In this case the “LEAVE($k_3$)” sent by $k_1$ and the “LEAVE($k_3$)” sent by $k_2$ are ordered and acknowledged. By FIFO “START\_REMOVE($k_1$)” arrives before “ACK\_LEAVE” at $k_2$.

Thus, logically we can consider that each leave is managed sequentially (the first basic step undertaken by $k_1$ is handled before the second step undertaken by $k_2$). Thus, from the time in which $k_2$ restarts the basic step to the time in which $k_2$ receives the “ACK\_LEAVE” message (from $k_0$ and $k_3$), all $k_m$ with $\ell \leq m \leq 1$ are strong connected, and so on until $k_{t+1}$ becomes a child of $k_0$ and $k_0$ the parent of $k_{t+1}$ after the leaves of all processes $k_m$ with $\ell \leq m \leq 1$. Then all $k_m$ processes with $\ell < m \leq d$ are strong connected and belong to $K(t')$ for each $t < t' < t''$ where $t''$ is the time in which $k_t$ leaves.

**case3.** Suppose that not all processes $k_m$ with $\ell \leq m \leq 1$ concurrently trigger a transition [in/leaving] in $B(t)$, but only a number $l < m$. In this case the branch $B(t)$ can be subdivided in sub-chains in which the extremes are processes not leaving. Then this case can be reduced to the previous ones.
Theorem 2.2.4 (Active Strong Connectivity). At any time \( t \),
\[
\forall i, j : \langle i, i.X \rangle, \langle j, j.X \rangle \in O(t) \Rightarrow (i \rightarrow_t j) \land (j \rightarrow_t i)
\]

**Proof.** By Lemma 2.2.3, the Theorem follows.

Theorem 2.2.5 (Global Progress). If \( \exists i \in \Pi \) that executes the join\(_i\) action at time \( t \), then \( \exists t', j : t' > t \land \langle j, j.X \rangle \in O(t') \land \langle j, j.X \rangle \notin O(t) \)

**Proof.** By Theorem 2.1.5

Finite Memory for Neighbor Variables. Adopting the technique used in Protocol C, it is possible to remove all stable false children. Moreover, thanks to flooding it is even possible eventually remove all old neighbors, i.e. neighbors which have left the overlay. Let us remind that this collection of old neighbors is done by assuming FIFO channels. By finite and unbounded concurrency, the memory required to store neighbor variables is finite and unbounded.

2.3 Connectivity Maintenance and Broadcast Problems

Dynamic Reliable Broadcast. Reliable Broadcast is a well-known communication primitive in the context of static systems [24] and has to ensure a certain reliability on the message delivery. Of course in a dynamic system the reliability of a broadcast has a weaker sense than the one defined in a static system, because of possible leaves and joins. Thus, we introduce the dynamic reliable broadcast (DRB) communication primitive, with two different specifications. The first one is the following:

**DRB1 (Validity).** If, at a time moment \( t \), a stable vertex \( i \) broadcasts a message \( m \), then \( m \) is eventually delivered by each stable vertex \( j \) such that \( j \in O(t) \).

**DRB2 (No creation).** If a vertex \( i \) delivers a message \( m \), then \( m \) was previously broadcast by some vertex.

**DRB3 (No duplication).** No message is delivered more than once.

The second one, namely **DRB’** maintains No Creation and No Duplication but strengthens Validity in Active Validity:

**DRB’1 (Active Validity).** If, at a time moment \( t \), a correct vertex \( i \) broadcasts a message \( m \), then \( m \) is eventually delivered by each stable vertex \( j \) such that \( j \in O(t) \).

Relations between Connectivity Maintenance and DRB/DRB’.

**Theorem 2.3.1.** In any model, \( AS \geq DRB’ \), i.e., if \( AS \) can be solved then \( DRB’ \) can be solved.
Proof. Let $i$ and $j$ be two vertices belonging to the overlay at a time moment $t$, and suppose that $j$ is stable. Due to AS at any time there exists a strong connection between $i$ and $j$. Consider the following simple flooding protocol: $i$ sends $m$ to each of its neighbors; when a process receives $m$ for the first time, it forwards $m$ to each of its neighbors except to the sender of $m$. Due to AS, $m$ will reach at least the stable vertices (and may be some other processes). This ensures DRB’1. The two other properties are obvious. □ □

Theorem 2.3.2. In any model, we have $ES \geq DRB$, i.e., if $ES$ can be solved then $DRB$ can be solved.

Proof. The stable vertex repeatedly retransmits the message $m$. Eventually, connectivity is restored. The Broadcast described in Theorem 2.3.1 guarantees that eventually $m$ will reach all stable vertices. □

Theorem 2.3.3. If joins eventually subside, we have $DRB \geq ES$, i.e., if $DRB$ can be solved then $ES$ can be solved.

Proof. Any process $i$ enters the overlay setting $i.X = \{i\}$ and broadcasts a message through $DRB$ with its identifier. Eventually each stable vertex includes in its neighbor variables other stable vertices. If joins eventually subside, the number of stable vertices is finite. It means that eventually all vertices in the overlay are strong connected. □

Corollary 2.3.4. In a finite arrival model, $DRB$ and $ES$ are equivalent.
Chapter 3

Connectivity Maintenance
Protocols Handling Failures

Connectivity maintenance under (i) asynchrony, (ii) continuous arrival/departure of nodes and, (iii) presence of failures, is a very difficult task. In the previous chapter an approach to face with continuous arrival/departure of correct nodes and asynchrony has been shown. It has been proved that, in absence of failures, it is possible to guarantee active (weak/strong) connectivity, i.e. connectivity at any time. Before presenting an approach handling failures, we would briefly examine some key concepts, namely, (1) the concept of failure in a dynamic system characterized by unbounded concurrency and an infinite arrival model, (2) the limitations of deterministically maintaining connectivity in such a model, (3) the meaning of a voluntary leave.

3.1 Key Concepts

3.1.1 A Failure in a Continuous Evolving System

The first difficulty in designing protocols handling failures in a dynamic system in which (i) the number of processes is unknown and (ii) infinite processes may participate (in infinite runs) to the distributed computation, is defining what a failure means in these systems. In static systems a crucial parameter is the maximal number of processes that can be faulty. This number defines a threshold \( f \) of failures such that in any run in which the threshold holds, a (correct) protocol is supposed to behave as expected. In a dynamic system this prefixed threshold \( f \) is meaningless [39]. In our model, the number of processes that can eventually participate to the distribute computation is infinite, thus fixing a threshold of failures to \( f \) means tolerating \( f \) failures out an infinite number of processes \( n \). Then, having a protocol that behaves correctly only when the total number of failures is maximum \( f \), would be so useless to be discharged in practice.

\[ \text{Even with the finite arrival model [38] and an unbounded number of processes in any run, the fraction of faulty nodes is not deterministically defined.} \]
Any protocol should be designed having in mind that no threshold will hold.

3.1.2 Limitations of Maintaining Connectivity

Under hypothesis of unbounded concurrency and occurrence of failures, it is not possible guaranteeing Active connectivity. Guaranteeing connectivity at any time means tolerating a number of leaves/failures equal to the number of vertices at any time. This is could be possible if the maintenance protocol were able to maintaining a clique at any time. Unfortunately, the impossibility of maintaining a clique (see Impossibility 1.3.7) at any time under concurrency holds. An overlay topology is $k$-connected if it is able to remain connected after the simultaneous deletion of any subset of nodes with cardinality fewer that $k$ [17]. If a protocol were able to maintain a $k$-connected graph at any time, then Active connectivity would be deterministically assured if at any time at most $f = k$ failures occur. However maintaining a $k$-connectivity at any time means that if a failure occurs it is immediately repaired. Even though it is possible with voluntary leaves, it cannot be assumed with failures. What it is deterministically possible, however, is maintaining a $k$-connected overlay topology until $k$ failures occur \(^2\). As said in Section 3.1.1, in our model, a protocol that behaves correctly (by assuring active connectivity) only until a prefixed threshold of failures $k$ holds, is useless. In practice, the maintenance of active connectivity should be so limited to be meaningless. Thus, if an unbounded number of failures may occur at any time (an infinite number in any infinite run) partition cannot be avoided. Then, any connectivity protocol in our model must be able to recover from partitions.

To assure Eventual connectivity, however, partitions must subside. Partitions may subside only if the number of failure is finite. The number of failures is finite in any run in which the number of processes that adds to the overlay is finite. In these runs, we say that joins eventually subside. Only for these runs (as we see later) it is possible guaranteeing eventual connectivity. In other terms the problem is solvable only assuming the finite-arrival model [38].

3.1.3 The Meaning of a Leave

In any model, a faulty process is a “bad” process, i.e. a process that departs from the expected behavior. In our model, a leaving process cannot be considered a bad process. A leave, like a join, is a provided action that a process can undertake. In other words all correct processes both stable and transient are “good” processes.

Under the optimistic assumption that most of the processes participating the distributed computation are good ones, i.e. leaves occur more frequently than failures, the design of a connectivity protocol should consider that the main source of non-determinism that threatens connectivity is the continuous arrival/departure of “good nodes”. This consideration boils down in handling leaves in a different way with respect to failures and trying to not partition an overlay of good processes.

\(^2\)Depending on the structure of the overlay, it is possible to have run in which active connectivity is assured even when more than $k$ failure occurs.
Unfortunately, as a failure is unavoidable, the protocol should cope even with bad processes. However, the handling of bad processes may be reactive and lazy, i.e. a detection mechanism detects the failure (that can lead even to a partition) and recovery mechanisms are provided.

3.2 Implementing Eventual Strong and Weak Connectivity

From the above considerations, the presented approach handles leaves actively, and thanks to a failure detection mechanism handles failures by providing restoring mechanisms. The solutions proposed guarantee eventual connectivity during periods in which the overlay stabilizes. In such periods, the protocol provides an overlay structure 1-connected. As said in Section 3.1.2 providing an overlay structure showing a $k$−connectivity with $k > 1$ does not add any sensible value in terms of deterministic guarantees. On the other hand, the maintenance of a more complex and redundant structure would require a not-negligible extra-effort and overhead.

In Section 3.2.1 we present a protocol which is an extension of the Basic Protocol of Section 2.1.1 handling failures. This protocol (as all protocols previously presented) does not address scalability issues. Techniques to keep scalability are discussed in Section 3.3 and a protocol is presented.

3.2.1 A Basic Protocol Variant Handling Failures

The presented protocol copes with the presence of failures by exploiting a failure detection mechanism. In particular, each vertex $i$ monitors its parent. If this parent seems failed, $i$ is in charge to find another live parent. We assume that this search is led exploiting the contact() function again. During the search the vertex enters a restoring state and maintains the suspected parent in its parent variable until a new parent is found. Children are not monitored. This brings the immediate advantage to maintain strong connection during a restoring state due to a false suspicion since the parent of $i$ (falsely suspected) does not accidentally remove $i$ from its children variable. Moreover, while in restoring state, $i$ maintains all its children. Thus, assuming no failure (and no leave) occurs concurrently to the restoring of $i$, we have that (i) if restoring is due to a false suspicion the (strong) connectivity of the overlay is not lost and (ii) if the restoring is due to a real failure, at the end of the restoring phase $i$ has reconnected all its sub-tree.

To safely reconnect the $i$’s sub-tree, $i$ selects among contacts, provided by contact() during restoring, only vertices not belonging to its sub-tree, otherwise a cycle arises (a cycle prevents the serialization of leaves).

Note that the protocol may converge easily to a small set of hubs during asynchrony (false suspicions) periods. In fact, false suspicions increases the number of concurrent processes that invokes the contact() function at a given point of time. At this point the contact() function can select a contact among a few vertices still in the overlay. Mechanisms to rearrange the topology during stabilization periods in the overlay (in terms of both asynchrony and overlay changes) may be pursued but are out of the scope of this dissertation.
Let us remark that this protocol is designed with the optimistic assumption that leaves are many more than failures. In this case by adopting an active approach for leaving and by using large time-outs for fault-detection, the protocol assures strong connectivity most of the time (weak connectivity during the overlay updating due to leaves).

The protocol presented in Fig. 3.2 uses the same neighbor variables used by the Basic Protocol seen in Section 2.1.1. Each process \( i \) can access to a \( FD_i \) local module returning a list of suspected faulty processes. Since \( out \) processes are not concerned with the protocol (they do not respond to protocol messages), \( FD_i \) manages \( out \) processes like faulty ones.

Initially, the transition \( joining/in \) brings the process \( i \) to select one parent, namely \( j \), which includes \( i \) as one of its children. When \( i \) starts to belong to the overlay (line 8), thanks to lines 27-28, the previous selected parent \( j \) is monitored. If the failure detector reports the parent \( j \) as suspected, then the variable degree turns to 0 and the process \( i \) switches in a restoring state (lines 29-30). Note that now \( i \) is in a restoring state and continues to belong to the overlay as parent is still set to \( j \) and \( i \) could have even some child. However, if the suspect is true, \( i \) is disconnected while it is in the restoring state. Then, now \( i \) is in charge to restore the overlay by connecting its sub-tree to a new live parent. To avoid cycles each process maintains an additional variable \( rank \) that gives an indication of the position of the vertex in the overlay. The value of \( rank \) is defined during the [\( joining/in \)] transition and it is never modified. The root \( r \) has \( r.rank = 0 \). If a process \( i \) joins through a process \( j \), \( i.rank \) will be set to \( j.rank + 1 \). When a process \( i \) turns in a restoring state, then it avoids to select as parent any parent \( k \) with \( k.rank \geq i.rank \).

In Fig. 3.1 a possible overlay evolution and the use of ranks is shown. After a growing phase in which only joins occur (Fig. 3.1(a)), a leaving phase in which only leaves occur is shown (Fig. 3.1(b)). Then a failure and the corresponding restoring is described in Fig. 3.1(c) and Fig. 3.1(d).

Anytime \( i \) switches in a restoring state, if \( i \) never completes lines 4-8, \( ES/EW \) will be violated. For this reason the protocol is based on the following assumptions:

**Assumption 3.2.1.** The contact() function eventually returns a stable vertex with \( s = in \) and \( rank = 0 \)

**Assumption 3.2.2.** The overlay \( O(t_0) \) = \( \{(r,(r.parent = r,r.children = NIL))\} \) such that \( r \) is stable, \( s = in \) and \( rank = 0 \).

These assumptions assure the completion of the transition \[restoring/in\]. Practically, the contact() function should eventually return to a restoring process \( i \) a stable process which does not belong to the sub-tree of \( i \).

Let us remark that in presence of failures Assumption 3.2.2 is necessary to preserve connectivity (while in absence of failure assures only termination). In case of the root failure, all processes with rank 1 would remain disconnected as no vertex with lower rank there exists. The issue of a single-point of failure arises, then this assumption may appear very strong. It is, however, practically satisfied by every P2P overlays relying on a set of super-peers like Kazaa [2]. In a pure peer-to-peer model, in which super-peers are not present, recent studies have revealed the presence of a stable component inside the network.
Mechanisms to select the root among these stable peers may be pursued. From a theoretically point of view, Assumption 3.2.2 strengthens the non-determinism pointing out the need of an overlay component not affected by non-deterministic behaviors.

To assure that a broken overlay will be eventually restored even the following assumption must hold:

**Assumption 3.2.3.** There exists a time $t$ after which a faulty vertex is permanently suspected by every correct process.

Even if each vertex $i$ restores the overlay, eventual connectivity (both EW and ES) is assured only if any vertex stops to switch on the restoring state. This also implies that eventual may hold connectivity only if joins (of new processes) eventually subside, since each new added vertex maybe faulty and then the overlay may need to be eventually restored again.
connectivity Management:

```plaintext
1 var : parent_change := ⊤ Boolean;
2 degree := null {0, 1, null};
3 rank := ∞ {null} ∪ Integer;

4 when (s = joining ∨ s = restoring) do
5 a := contact(); send ["JOIN", rank] to a;
6 wait until (degree = 1 ∨ a ∈ FDi ∨ receive ["RETRY"] from a)
7 when (receive ["ACK_JOIN", r] from a) do
8 parent := a; degree := 1;
9 if (rank = ∞)
10 then rank = r + 1;
11 when ((s = leaving) and (parent_change)) do
12 send ["LEAVE" (children)] to parent;
13 parent_change := ⊥;
14 when (receive ["ACK_LEAVE"] from a = parent) do
15 parent := nil; ∀k ∈ children := nil; s := out;
16 when ((receive ["JOIN",r"] from j)) do
17 if ((s = in) ∧ (rank < r))
18 then children := children ∪ {j};
19 send ["ACK_JOIN",rank] to j
20 else send ["RETRY"] to j
21 when ((receive ["LEAVE"(j,children)] from j ∈ children) ∧ (s = in)) do
22 children := children ∪ j.children − {j};
23 send ["NEW_PARENT"(i)] to each j.children; send ["ACK_LEAVE"] to j;
24 when (receive ["NEW_PARENT"(j)] from a) do
25 parent := j;
26 parent_change := ⊤; degree := degree + 1;
27 while parent ≠ nil do
28 degree := |parent| − |parent ∩ FDi|;
29 when (degree = 0) do
30 s := restoring;
31 when (degree = 1) do
32 s := in;
```

Figure 3.2: A Basic Protocol Variant Handling Failures at $i$

**Proof of the Protocol**

**Lemma 3.2.4.** Any transition [restoring/in] completes in finite time

*Proof. (Sketch) By Assumption 3.2.2, $i$ has a rank variable such that rank $> 1$. By Assumption 3.2.1 eventually a contact sends an “ACK_JOIN” to the restoring process $i$ (lines 16–19).*  

**Lemma 3.2.5.** At any time the graph defined by parent variables does not contain cycles.
Proof. (Sketch) Let us consider only the connections defined by parent variables (no child is considered). A cycle there exists iff \( i \leftrightarrow j \).

Base case. No cycle of length \( \ell = 0 \) are ever created. A cycle of length 0 arises iff \( i \leftrightarrow j \) with \( i.parent = j \) and \( j.parent = i \). This implies that there exists a time \( t' \) in which \( i.parent \) has been set to \( j \) and a time \( t'' \) in which \( j.parent \) has been set to \( i \). Without loss of generality, suppose that \( t' \leq t'' \). Three subcases:

- case1(a) \( i.parent \) has been set to \( j \), upon the \( i \)'s join. From the pseudo-code (line 10) immediately follows that \( i.rank > j.rank \).
- case1(b) \( i.parent \) has been set to \( j \), upon the \( i \)'s restore. From the pseudo-code (line 17) immediately follows that \( i.rank > j.rank \).
- case1(c) \( i.parent \) has been set to \( j \), upon the \( i \)'s receiving of a “NEW_PARENT” from \( j \). In this case \( i \) has been inherited by \( j \), after the leaves of some vertices, namely, \( k, k',...k'' \) that before \( t' \) have connected \( i \) and \( j \). Without loss of generality let us suppose that \( k, k',...k'' \) had established their connections in the first two cases (case1(a) and case1(b)). This implies that: \( j.rank < k.rank < k'.rank < ...k''.rank < i.rank \). Then \( i.rank > j.rank \).

Now, if a time \( t'' \) \( j \) sets \( j.parent \) to \( i \) from the above cases results that \( j.rank > i.rank \). Thus an absurd arises.

Inductive Step. No cycle of length \( \ell = l \) are ever created. This implies that any branch of length \( l+2 \), \( \{ j, k_1, ..., k_i, i' \} \) with \( i'.parent = k_i.parent = k_{i-1}, ..., k_2.parent = k_1, k_1.parent = j \) has \( i'.rank > k_i.rank \implies k_i.rank - 1, ..., k_2.rank > k_1.rank > j.rank \).

Now let us consider the branch: \( \{ j, k_1, ..., k_i, i' \} \) with \( i.parent = i' \). As in the base case \( i \) is connected to \( i' \) with \( i.rank > i'.rank \). This implies that \( j.rank < i.rank \). Then \( j.parent \neq i \) otherwise \( j.rank > i.rank \) and an absurd arises.

\[ \square \]

Lemma 3.2.6. The overlay loses connectivity if and only if a non-leaf fails.

Proof. sketch (1) If a non-leaf the overlay loses connectivity. Obvious.

(2) If the overlay loses connectivity a non-leaf has failed.

If the overlay loses connectivity this means that there exists at least one pair of vertices \( i \) and \( j \) with no directed path in between. In absence of failures and false suspicions by the proof of the Basic Protocol, \( i \) and \( j \) always have a directed path in between. With false suspicions the protocol maintains weak connectivity: a child that suspects its parent maintains the connection with its parent and vice-versa, until a new parent is found (lines 29-30 and 7).

\[ \square \]

Theorem 3.2.7. If joins eventually subside at time \( t' \), there exists a time \( t \) after which weak connectivity holds.

Proof. (sketch) If joins subside a time \( t' \) then it is possible comprise in a finite set \( M \) the vertexes that can belong to \( O(t'') \) for each \( t'' > t' \). Let us consider \( M \) subdivided in three subsets \( F, T, S \). \( F \) are all faulty processes in \( M \), \( T \) all transient and correct processes and
all stable processes. By Lemma 3.2.6, at most $F$ faults disconnect the overlay. Without loss of generality, let us suppose that at time $t'$ Thanks to Assumption 3.2.3 all $F$ faults will be detected. Upon the detection by a vertex $i$ (transient or stable) all the sub-tree of $i$ defined by parent variables is eventually reconnected by Lemma 3.2.4. By Lemma 3.2.6, after all $F$ faults are restored at time $t$, the overlay $O(t)$ is forever weak connected and comprises all processes in $S$ and processes in a possible empty set $T' \subseteq T$.

**Corollary 3.2.8.** If joins and leaves eventually subside strong connectivity eventually holds.

*Proof.* The corollary follows from 2.1.9 and Theorem 3.2.7.

**Memory for Neighbor Variables and False Suspicions.** In the presented protocol children are not monitored, to avoid that maintaining connectivity depends on the underlying network behavior. As a consequence the memory required to store neighbor variables is infinite. On the contrary, guaranteeing finite memory while assuring eventual connectivity requires the presence of an eventually perfect failure detector, i.e. eventual connectivity can be assured only if the overlay stabilizes in terms of joins/leaves and of the underlying network behavior.

From a practical point, children may be monitored using time-outs extremely large. This should make false suspicions very unlikely and gives the possibility of removing faulty children.

### 3.3 Techniques to Keep Scalability

Until now we have addressed connectivity considering the possibility that any node is always able to accept new connections. However, to maintain the scalability of the overlay structure the degree per-node should be well-balanced (not necessarily bounded).

The maximum degree per node, i.e. number of children (indegree) + number of parents (outdegree), may be predefined or mechanisms to adapt the degree to the total number of nodes may be figured out. First of all, if the number of accepted children (indegree) is defined and never changes, it means that when a node $i$ leaves, some of its child may be not inherited by the $i$’s parent. In this case the non-accepted vertices should enter in a restoring state.

An adaptive mechanism for resizing partial views can be the same used by Scamp [21]. An indirection mechanism (this mechanism is explained in Section 4.1) may be pursued to allow a joining node to choose its contact uniformly at random, starting from a node with rank 0. Each node $i$ accepts with a probability equal to $1/(|\text{children}| + 1)$ both (i) a newly joining process $j$ (in this case $i$ is the contact of $j$) and (ii) a process $k$ to inherit from a leaving child. If $i$ does not accept a newly node $j$, $j$ is in charge to retransmit the join message (to a node uniformly chosen at random). If $i$ does not accept a vertex $k$, $i$ sends a message to $k$ to warn it that its outdegree is being decreased. When $k$ knows that its outdegree has been decreased, it can re-join by entering in a restore state to find another parent.
Note that, the transition \([\text{joining/in}]\) completes only if the \(\text{contact}()\) function eventually returns a stable vertex able to accept new connections. In both cases considered above (prefixed maximum out-degree and adaptive out-degree) this situation surely occurs when the overlay stabilizes and the joining node can retrieve through the \(\text{contact}()\) function a stable leaf of the tree.

The transition \([\text{restoring/in}]\) can complete under the condition for the completion of \([\text{joining/in}]\) transition provided that the restoring node is not responsible for reconnecting all its own sub-tree but only itself. As seen in the previous protocol, reconnecting the sub-tree means finding a parent that is not in the sub-tree. This situation was solved by using ranks. However, if the restoring node is prevented to re-connect to all vertices with rank lower than its rank, then the reconnection is not assured.

Fig. 3.3 shows a scenario with maximum out-degree equal to 3. In this case the failure of a vertex of rank 1, prevents at least two children with rank 2 from restoring. In case of adaptive degree, the restoring is uncertain, unless the contacted node are leaves of the tree. However, even in this case the leaves of the tree have rank not lower than the restoring nodes.

![Figure 3.3: A Possible Topology Preventing Reconnection](image)

To solve this problem, any vertex that suspects a parent failure goes in \(\text{restoring}\) state but sets its children to nil. In practice, any disconnected node forces its children to restore by themselves. Note that, if a failing node has a deep sub-tree, a failure brings to a cascade of restoring nodes.

In the following a simple protocol with a prefixed maximum out-degree is shown.

### 3.3.1 A Basic Protocol Variant maintaining Maximum Degree \(D\)

The protocol is shown in Fig 3.4. Each process \(i\) has \(i.X\) composed by a pair \((i.parent, i.children)\) where \(i.children\) is composed by at most \(D\) elements.

The variable \(\text{outdegree}\) has the same sense of the variable degree seen in the previous protocol. The variable \(\text{indegree}\) counts the children number.
connectivityManagement_
i

52

var: parent_change := ⊤ Boolean;
outdegree := null \{0,1,null\};
indegree := 0; Integer;
constant : D =

5 when (s = joining ∨ s = restoring) do
6 a := contact(); send ["JOIN"] to a;
7 wait until (outdegree = 1 ∨ a ∈ FD, ∨ receive ["RETRY"] from a);
8 a := nil; % This line disables the successive receiving of "ACK_JOIN" messages, until a is set again %
9 when (receive ["ACK_JOIN"] from a) do
10 parent := a; outdegree := 1;

11 when ((s = leaving) and (parent_change)) do
12 send ["LEAVE"(children)] to parent;
13 parent_change := ⊥;
14 when (receive ["ACK_LEAVE"] from a = parent) do
15 parent := nil; ∀k ∈ children := nil; s := out;

16 when ((receive ["JOIN"] from j)) do
17 if ((s = in) ∧ indegree < D)
18 then children := children ∪ \{j\};
indegree + +;
19 else send ["ACK_JOIN"] to j
20 when ((receive ["LEAVE"(newchildren)] from j ∈ children) ∧ (s = in)) do
21 while indegree ≤ D do
22 ∀k ∈ newchildren ∧ k ∉ inherited : inherited := inherited ∪ \{k\};
indegree + +;
23 children := children \ inherited − \{j\};
24 send ["NEW_PARENT"(i)] to each inherited;
25 send ["NO_PARENT"(i)] to ∀k ∉ newchildren \ inherited;
26 send ["ACK_LEAVE"] to j;
27 inherited := ∅;
28 when (receive ["NO_PARENT"(j)] from a) do
29 outdegree := outdegree − 1;
30 when (receive ["NEW_PARENT"(j)] from a) do
31 parents := parents ∪ j; parent_change := ⊤;
32 while parent ≠ nil do
33 outdegree := |parents| − |parents ∧ FD,|;
34 when (outdegree = 0) do
35 children := nil; s := restoring;
36 when (outdegree = 1) do
37 s := in;

Figure 3.4: A Protocol with Maximum Degree D at i
The protocol is substantially an exemplified version of the previous one. In this case rank variables do not concern. Simply, when a process $i$ suspects to be disconnected sets its children to nil and $s$ variable to restoring (lines 36,37). Eventually the $FD$ of each dismissed children will report $i$ (by completeness).

The protocol works on the following assumption:

**Assumption 3.3.1.** *The contact() function eventually returns a stable vertex with indegree $< D$.***

Assumption 3.3.1 may be implemented only if joins eventually subside and faulty children are eventually removed. In this case the overlay stabilizes and it is possible to get a stable vertex able to accept new children (e.g. a leaf). Practically, each restoring node has to find a vertex able to accept it.

Then this protocol has to rely on the completeness of a failure detector and it needs to monitor children. This implies that Eventual connectivity (both EW and ES) can be assured only if (i) all overlay changes eventually subside (joins and leaves) and (ii) the underlying system eventually behaves as a synchronous one.

It is worth to point-out what would mean assuming a perfect failure detection for children. If no false suspicions involves children the following scenarios may occur:

**scenario 1.** Every non-inherited child $i$, (even though poses parent to the non-accepting parent $j$ until a new parent is found) can lose the connection to the overlay in the case $j$ is correct but leaves during the restoring of $i$ ($j$ does not give a notice to $i$ upon leaving because $i$ is not one of $j$’s children)\(^3\).

**scenario 2.** Every suspicion (related to a parent) may bring to lose connectivity if leaves can occur. Even if during the restoring state parent is not nil, when the parent variable of $i$, $i$.parent, points to a restoring parent $j$, $i$ may lose the connectivity with $j$, if $j$, once restored, leaves the overlay ($j$ does not give a notice to $i$ upon leaving because $i$ is not one of $j$’s children).

Then if false suspicions of children never occur, from the protocol, if $i$ restores when leaves already ceased, its children have parent $= i$ until they find another parent $j$. This implies that if $i$ restores due to a false suspicion related to a parent, weak connectivity (defined by parent variables) is not lost. On the other hand since any restoring node sets its children variable to nil, any restoring leads to lose strong connectivity. Then, the presented protocol is able to assure EW if joins and leaves and false suspicions of children eventually subside.

The presented protocol is able to assure ES if joins, leaves and (every) false suspicions eventually subside.

Practically, with extremely large time-outs for monitoring children and smaller time-outs (not too small in order to take advantage from active leaves) for monitoring parents, weak connectivity holds during periods in which the overlay stabilizes, while strong connectivity holds when the overlay stabilizes and the underlying system behaves as a synchronous one.

---

\(^3\)Observation 1 implies that the Basic Protocol extended to support bounded memory for neighbor variables in absence of failures (and in absence of failure detection mechanisms) it is not able to support active (weak) connectivity at any time.
Chapter 4

Performance Evaluation

The objective of this section is to compare and explore the behavior of probabilistic connectivity protocols supporting large scale information dissemination and deterministic connectivity protocols under continuous arrival/departure (churn) of nodes (processes). As representant of probabilistic connectivity protocols we chose SCAMP [21]. The reason behind this choice is twofold: (i) SCAMP is able to adapt the overlay structure to a changing number of nodes. In our model the number of nodes in the overlay is not predefined, so this adaptability characteristic is fundamental, (ii) SCAMP uses an active approach for leaves. As representant of the deterministic connectivity protocols we choose the Basic Protocol, in the following called DET, with a particular implementation of the contact() function. Let us remind that DET does not tolerate failures. However, the focus of these simulation is studying the resilience of these protocols under dynamics, i.e. the only source of non-determinism considered is churn. Thus, no failures is simulated not to bias results.

Let us remark that the aim of SCAMP is the maintenance of an overlay structure able to support information diffusion in reliable and scalable manner. More specifically, SCAMP supports a gossip-based information diffusion in which message redundancy cope with random network failures (reliability) and operational overhead grows only with the logarithm of the network size (scalability). The overlay structure built and maintained converges to a random graph. The aim of deterministic connectivity protocols is slightly different. Every deterministic connectivity protocol, as the ones presented in these thesis, is designed to maintain connectivity (with different specifications) in a deterministic way. The scalability issue is not directly addressed. Moreover, addressing scalability issue boils down to maintain a particular overlay structure (e.g. a tree with bounded degree). As showed in previous sections, the maintenance of a particular structure and not only of a connected structure, is possible only in stability periods (periods without churn) and partition is not avoided.

The dimensions taken into account in the simulation are reliability and scalability. Reliability and scalability are intended as the dimensions characterizing an application-level multicast running over the overlay. Both dimensions depend on the overlay structure the connectivity protocol is able to maintain: the capacity of keeping the overlay connected affects reliability and the way in which the overlay is connected affects scalability.
In setting up the simulation experiments, we faced the difficulty to come up with meaningful measures in such a continuously evolving systems. For example, to evaluate the reliability of the overlay networks, simulation studies often consider the proportion of the nodes reached by an application-level multicast. But in the presence of high dynamics (with high rates of joins and leaves), it is hard to define “which nodes must be considered the intended receivers of a given message”. We, therefore, used a well-known methodology which divides the simulation time into four intervals: a bootstrapping interval, a perturbation interval, a transitory interval and a measurement interval. Bootstrapping is the interval of time in which only joins are allowed. During the perturbation interval all joins and leaves occur at a given churn rate. At the end of the perturbation period, the churn terminates. After a transitory interval in which the system stabilizes, the measurement interval starts.

Using this methodology, experiments confirm that SCAMP suffers a poor reliability under churn while it scales for any churn rates. In particular under a churn rate equal to 1 membership change (a join or a leave) per second the proportion of nodes reached by the multicast is the 80%. However, the node degree (for every node) remains always equal to the logarithm of the number of nodes in the overlay. The analysis of the DET shows that the overlay remains connected for any churn rate. This form of reliability compromises scalability. In particular, to obtain a scalable structure, i.e. a tree well-balanced, the join/leave latency increases as the churn rate increase, becoming unpredictable (these latencies are negligible in SCAMP). On the contrary, to obtain low latencies for joining and leaving, the overlay structure converges to a star as much as the churn rate increases, thus resulting in an overloading of one node completely. This means that scalability is compromised in any case.

4.1 The SCAMP Probabilistic Protocol[21]

Scamp (Scalable Membership Protocol) is a connectivity protocol designed to support gossip-based dissemination, which is fully decentralized and provides each node with a partial view of the membership (neighbors). It is adaptive w.r.t. a-priori unknown size of the overlay, by resizing partial views when necessary. This protocol consists of mechanisms for nodes to subscribe (join) and unsubscribe (leave), and for nodes to detect and recover from isolation. The following is a brief description of these mechanisms.

**Data Structures.** Each node maintains two lists, a PartialView of nodes it sends gossip messages to, and an InView of nodes that it receives gossip messages from, namely nodes that contain its node-id in their partial views.

**Subscription Algorithm.** New nodes join by sending a subscription request to an arbitrary member, called a contact. They start with a PartialView consisting of just their contact. When a node receives a new subscription request, it forwards the new node-id to all nodes of its own PartialView. It also creates c additional copies of the new subscription
(c is a design parameter that determines the proportion of failures tolerated) and forwards them to randomly chosen nodes in its PartialView. When a node receives a forwarded subscription, provided the subscription is not already present in its PartialView, it integrates the new subscriber in its PartialView with a probability \( p = 1/(1 + \text{sizeof PartialView}_n) \). If it doesn’t keep the new subscriber, it forwards the subscription to a node randomly chosen from its PartialView. If a node \( i \) decides to keep the subscription of node \( j \), it places the id of node \( j \) in its PartialView. It also sends a message to node \( j \) telling it to keep the node-id of \( i \) in its InView.

**Unsubscription Algorithm.** Assume the unsubscribing node has ordered the id’s in its PartialView as \( i(1), i(2), \ldots, i(l) \) and the id’s in its InView as \( j(1), j(2), \ldots, j(l) \). The unsubscribing node will then inform nodes \( j(1), j(2), \ldots, j(l-c-1) \) to replace its id with \( i(1), i(2), \ldots, i(l-c-1) \) respectively (wrapping around if \( (l-c-1) > l \)). It will inform nodes \( j(l-c), \ldots, j(l) \) to remove it from their list but without replacing it by any node id. In the unlikely event that this mechanism requires a node to maintain multiple copies of a node-id, or to maintain its own id in its partial view, the corresponding id is simply deleted. Note that this protocol remains local and only the unsubscribing node and its direct neighbors in the graph are involved in the unsubscription process.

**Recovery from isolation.** A node becomes isolated from the graph when all nodes containing its identifier in their PartialViews have either failed or left. In order to reconnect such nodes, a heartbeat mechanism is used. Each node periodically sends heartbeat messages to the nodes in its PartialView. A node that has not received any heartbeat message in a long time resubscribes through an arbitrary node in its PartialView.

**Indirection.** This mechanism lets new subscriptions to be targeted uniformly at existing nodes in the overlay. This is done by forwarding the newcomer’s subscription request to a node that is chosen approximately at random among existing members. The interested reader may refer to [21] for further details. The indirection mechanism consists of two parts, a forwarding rule and a stopping rule. The stopping rule determines whether a node receiving a forwarded subscription request is going to treat it (and act as the contact). If the node decides not to treat it, the forwarding rule specifies to which of its neighbors it should forward the subscription request.

**Forwarding rule:** The forwarding step requires node \( p_i \) to forward the request to a node \( p_j \) in its partial view with probability \( w_{ij} \) (weights of the links). These weights should form a non-negative matrix irreducible and doubly stochastic. In this way the Markov chain associated with this transition matrix has the uniform distribution as its unique steady state distribution, i.e. the random walk associated with the forwarding rule is the uniform distribution on all nodes. Details on the definition and dynamic updating of these weights can be found in [21].

**Stopping rule:** When a new subscription is received by a node \( p_i \), it associates a counter with the subscription, initialized to a value \( n_i \) proportional to the size of node \( p_i \)’s partial view. It then forwards the subscription along with the counter to one of the members (decided by forwarding rule) of its partial view. Each node receiving a forwarded subscription decrements the counter and continues forwarding it. When the counter is zero, the node receiving the subscription is treated as its contact in the manner described by the
subscription protocol.

**Lease mechanism.** Each subscription has been given a finite lifetime called its lease. When a subscription expires, every node holding it in its PartialView removes it from the PartialView. Each node re-subscribes at the time that its subscription expires. Nodes re-subscribe to a node chosen randomly from their PartialView. Re-subscriptions differ from ordinary subscriptions in that the partial view of a re-subscribing node is not modified. This technique helps to rebalance the size of partial views across nodes.

**The determination of the contacts in SCAMP.** For SCAMP the contact is an arbitrary node chosen among current nodes of the overlay. In order to implement the retrieval of the contact we have provided an extra mechanism in which the joining node broadcasts a message to II and chooses its contact inside the list of non-out nodes that have replied to the broadcast. Clearly, for high churn rates this node may choose a contact that has left the overlay immediately after the reply. Note that for SCAMP once a subscription is sent, the node is logically a vertex of the overlay. Then, an out contact is a real problem that affects reliability. Note that even the indirection mechanism does not solve this problem as it is a mechanism that works well in fairly static systems [21].

### 4.2 The DET Protocol

The DET protocol is a variant of the Basic Protocol in which the state of a process \( i \) includes an integer variable \( \text{rank}_i \) which gives an indication of the position of \( i \) in the overlay.

**Initialization of the Overlay.** The overlay is initialized by an initiator \( r \), i.e. \( O(t_0) = (r, (r.parent = r, \forall x \in r.children, x := \text{nil})) \land r.state = \text{in} \land \text{rank} = 0. \) \( r \) is assumed to be a stable process.

**Implementation of the contact() function and Mechanism to Join.**

In this version of DET we use the following implementation of the contact() function: the contact function returns a list of contacts in which the node \( r \) with rank 0 is always comprised and the other nodes are arbitrary. The joining nodes \( i \) sends a JOIN message to contacts. Then \( i \) collects responses from contacts. The implementation provides that each contact responds to the JOIN message with its rank (all non-out processes do not send an acknowledgment but only an empty message). Then, \( i \) selects as its parent the node with the highest rank among the ones which acknowledged \( i \) and sets its own rank to the rank of the selected parent plus one. In the simulation we will vary the size of the contacts list. As expected, simulation results will show that the more the size of the contacts list, the more the probability to augment the height of the tree. Since all contacted in processes add the joining node as child even if this node will select only one parent among all contacts, extra-messages are needed to purge false children. Mechanisms to remove false children have been discussed in Section 2.1.3. In this simulation only stable false children are removed (children that arrives in the measurement interval) through the technique seen in protocol C (see Fig. 2.11). With this technique stable children are removed in the transitory interval completely.
**SCAMP vs DET.** In SCAMP when a node wants to join or leave, it simply sends a join or leave message and does not wait for acknowledgement messages. In DET whenever a node wants to join or leave the group, it has to wait for the acknowledgement message after sending the request. The advantage of this type of behavior is that it supports connectivity. But obviously this advantage is not without any cost. The cost incurred is the latency in joining and leaving the overlay. In this respect DET requires a more responsible node behavior than SCAMP when a node leaves the group. This may be a drawback in a peer-to-peer environment in which nodes are supposed to be selfish, i.e. they do something only if they gain something.

**4.3 Simulations**

Simulation is conducted by using Ns-2 simulator. The choice of Ns-2 was mainly due to the reason that we want to use the full protocol stack and test our protocol at the application level to have as far as possible real scenario of the working of the protocol including asynchrony and concurrency. But, also as remarked in [3], due to the exponential nature of the phenomena it was only possible to simulate for small view size and / or high churn rates.

Let us remark that in order to understand what is the real impact of dynamics (joins and leaves/sec) on these protocols we conducted simulations in which no failure is simulated but only joins and leaves.

**4.3.1 Simulation Framework**

Each simulation involves a global number of nodes $n_{tot}$. Each simulation is divided in four intervals: the bootstrap interval, the perturbation interval, the transitory interval and the measurement interval (see Fig.4.1).

![Figure 4.1: Time intervals of a simulation run](image)

*The bootstrap interval.* The bootstrap interval $\Delta_b$ is intended as the phase in which the group grows (until a desired value is reached) and no leave occurs. In the bootstrap interval the group starts at $t_0$ with $n_0$ bootstrap nodes. At the end of the bootstrap interval ($t_1$) the group contains $n_1$ nodes. This means that the membership changes in the bootstrap interval consist in $n_1$ joins.
The perturbation interval and the transitory interval. The perturbation interval $\Delta_p$ is intended as the interval in which all membership changes (joins and leaves) are injected in the system. The transitory interval $\Delta_t$ is intended as the interval in which all membership changes injected in the perturbation interval take effect. In each simulation the group starts this interval at $t_1$ with a number of nodes $n_1$ obtained after the bootstrapping it ends at $t_3$ with a number of nodes $n_f = \frac{1}{2}n_{tot}$. In the perturbation interval we have a total number of leaves equal to $\frac{1}{2}n_{tot}$ and a number of joins equal to $n_{tot} - n_1$.

The measurement interval. In the measurement interval $\Delta_m$ all measures are taken. In particular we test for both protocols (i) the proportion of nodes reached by a set of (data) messages sent by each node during the measurement interval, (ii) the average node degree and its distribution, where the node-degree is the number of active connections per-node. The first and second metrics are related to the level of reliability shown by the protocols. Moreover, the second metrics shows the overhead of the protocol. In the case of our protocol we also test the average latency of leaves, i.e. the average time between the leave invocation and the actual departure of the node from the group.

All measures are taken by varying the dynamics rate in the perturbation interval. To characterize the dynamics we use the churn rate metrics. The churn rate is the ratio between the number of membership changes, i.e. joins and leaves, and the duration of the perturbation interval. By considering a fixed number of joins and leaves, the churn rate varies by varying the duration of $\Delta_p$. In particular for each simulation $\Delta_p$ varies from 5 sec to 200 sec. Arrivals and up-times follow an exponential distribution.

Simulated Scenarios. All the following simulations have $n_{tot} = 160$. We have first compared the two protocols in a scenario in which no bootstrap occurs. Then we have also evaluated SCAMP in another scenario to study the impact of bootstrapping. Table 1 resumes the simulated scenarios. The scenario in which $\Delta_b > 0$ is subdivided in three scenarios. In particular, we have evaluated what is the SCAMP behavior when (i) $1/4n_{tot} = 40$ nodes enter the perturbation interval and all of them leave the group during the perturbation (0% of stable nodes), (ii) $1/4n_{tot} = 40$ nodes enter the perturbation interval and half of them leave the group during the perturbation (50% of stable nodes) and (iii) $1/4n_{tot} = 40$ nodes enter the perturbation interval and none of them leave the group during the perturbation (100% of stable nodes).

<table>
<thead>
<tr>
<th>$\Delta_b$</th>
<th>$n_0$</th>
<th>$n_1$</th>
<th>joins during $\Delta_p$</th>
<th>leaves during $\Delta_p$</th>
<th>$n_f$</th>
<th>% stable nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>160</td>
<td>80</td>
<td>80</td>
<td>-</td>
</tr>
<tr>
<td>$&gt;$ 0</td>
<td>1</td>
<td>40</td>
<td>120</td>
<td>80</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

In the first scenario and in the second scenario, at the beginning of $\Delta_b$, $\Delta_p$ respectively, the starting node has neighbor variables containing only itself.

1 A connection is said active when its ends are in the overlay.
Each point in the plots has been computed as an average of 40 simulation distinct runs. For each point all the results of these runs were within 4% each other, thus variance is not reported in the plots.

**Protocols parameters.** As no failure are simulated, we consider for SCAMP $c = 0$. Even if we do not consider failures we have implemented for SCAMP a heartbeat mechanism to avoid isolation due to leaves\(^2\). The heartbeat mechanism we figured out forces a node to re-subscribe if it has not received any heartbeat from its InView in 2.5 seconds. In some plot we have also implemented the lease mechanism for SCAMP. We set the lease duration equal to 50 seconds\(^3\).

### 4.3.2 Experimental Results with no Bootstrapping

**Evaluating Reliability of the Topology generated by SCAMP & DET.** In the measurement interval the overlay is frozen in a certain configuration.

![Figure 4.2: A comparison between SCAMP and DET after a $\Delta_p$ with churn rate equal to $240/\Delta_p$](image)

Thus, overlay vertices at the beginning of this interval remains in the overlay till the end of the simulation and no new node is added. The plot in Fig.4.2(a) shows that DET is able to guarantee that each message sent by a vertex in the measurement interval is delivered by every vertex independently of the churn rate suffered during the perturbation. This confirms that DET assures weak connectivity during the perturbation interval (no partition

---

\(^2\)Isolation may occur since contacts leave the system without giving a notice to nodes which joined through it.

\(^3\)This value could be chosen smaller or bigger, but a value of 50 secs allows a reasonable number of resubscriptions in the considered intervals both in terms of scalability and in terms of added reliability.
occurs) and that when leaves subside DET is able to assure strong connectivity, i.e. reliable communication.

On the other hand, SCAMP is sensitive to different churn rates suffered in the perturbation interval. In particular, only with churn rates lower than 1.5 membership changes (joins or leaves) per-second the proportion of nodes reached by a multicast is the 80\% of nodes. Plot in Fig.4.2(b) shows as the poor reliability of SCAMP is due to a small average degree (from 2.1 to 2.7). This degree is ever less than the threshold of $\log(n_f = 80) = 4.38$ to be reached for a successful working of SCAMP.

The average node-degree of DET only points out that the built topology is a tree, it has no direct relation with reliability. In the next paragraph we discuss scalability of DET considering the node-degree distribution.

**Evaluating Scalability of the Topology generated by SCAMP and DET.** To evaluate the scalability for SCAMP and DET it is necessary to examine the structure of the topology that they build.

![Figure 4.3: Degree Distribution for DET at $t_3$ for $|\text{contacts}| = n_{tot}$ and $|\text{contacts}| = n_{tot}/10$](image)

In particular for DET the size of the contacts list has a huge impact on the overlay topology since a small contacts list contributes to keep the message overhead small but the obtained topology converges to a star topology with the node of rank 0 in the middle. In the Fig. 4.3 plots showing the distribution of the node-degree in case of a contact list with size equal to $n_{tot}$ (Fig. 4.3(a)) and equal to $n_{tot}/10$ (Fig. 4.3(b)). Note that the size of the contact list is a predominant parameter with respect to the churn rate. In particular, if the contact list is small the topology converges to a star even for low churn rates (Fig. 4.3(b) curve for $\Delta_p = 200$sec). With a large contacts list the topology converges to a star (more properly, the topology shows a set of hubs) only for high churn rates (Fig. 4.3(a))
curve for $\Delta_p = 5\text{sec}$), but the tree become deeper for low churn rates (Fig. 4.3(a) curve for $\Delta_p = 200\text{sec}$).

Thus, it is confirmed for DET that the faster joins (a join takes a time equal to the maximum round-trip time between contacts links) the least scalable is the topology. Note that more sophisticated mechanisms to join may be pursued (e.g. an indirection mechanism like in SCAMP), to maintain a small contact list and a scalable generated topology at the same time. However, these mechanism with high churn rates may lead to unpredictable latency of join/leave operations.

On the contrary, SCAMP is always able to balance the degree for each node (see Fig. 4.4(b)) showing a great scalability. The churn rate impacts only on the average degree and in the distribution degree (see Fig. 4.4(a)) affecting reliability.

**Evaluating Leave Latency for DET.** We evaluate leave latency as the average time that passes from the invocation of a leave (the sending of an unsubscription message) to the actual departure of the node from the group (the receiving of an acknowledgement). Note that for each node of rank 1, this time is equal to the round-trip time on the link connecting the node with the node with rank 0. As the rank increases the latency may increase as well. In the worst case a node with rank $i$ may concurrently invoke its leave with all nodes with lower rank belonging to its branch. In this case the latency becomes proportional to the rank of a node. Three factors influence the latency of a leave (i) the depth of the tree (deeper trees bring higher latency), (ii) the rate of leaves (higher rates brings higher latency) and (iii) link delays. The first factor depends on the size of the contact list. In practice, with a contact list very small (as pointed out in the previous paragraph) the tree converges to a star. In this case the average latency is equal to the round-trip time on the link connecting the node with the node with rank 0. The third factor depends on the underlying network behavior, then it may unpredictable. To avoid that an unexpected
network behavior biases our analysis we consider (only for this particular evaluation) that all links have a RTT equal to 0.02ms. Then, we have chosen to evaluate the leave latency in the case in which (i) all node leaves the system at the same time, (ii) the contact list is very large (\(|contacts| = n_{tot}\)) and (iii) the churn rate is low (\(\Delta_p = 200s\)). In this way the tree is a branch and we can evaluate the worst case for leave latency but the best case for scalability of the topology.

In Fig. 4.5 the latency distribution and the number of conflicts for each node, i.e. the number of LEAVE messages received by a node when it was leaving, is shown.

In Fig. 4.5 the latency distribution and the number of conflicts for each node, i.e. the number of LEAVE messages received by a node when it was leaving, is shown.

The impact of the lease mechanism in SCAMP

The plots in Fig. 4.6 shows as the lease impacts the reliability of SCAMP under churn. In practice, the lease mechanism does not influence in the average the reliability of SCAMP. What the lease produces is a high clustering of the overlay, i.e. most of the nodes are very-well connected and some nodes are isolated. To point out this behavior see (i) Figure 4.7(a) in which the average number of isolated nodes is in SCAMP higher than in SCAMP without lease and (ii) Fig.4.7(b) in which not isolated nodes have an average degree higher than in the case of SCAMP without lease.

---

4 This value has been chosen so small only for convenience.

5 It means that during the perturbation there are two phases: a growing phase in which all joins occur and then a phase in which all leaves occur at the same time.
The reason underlying this behavior is that the lease mechanism forces even a connected node to re-subscribe contacting an arbitrary member of its partial view. With high churn this member may have already left. Even if the lease is repeated the same scenario may occur. On the other hand, for those nodes that find a node in the overlay upon the resubscription, there is a new dissemination in the system of their node identifier that enlarges partial views. It is clear that in fairly static systems (systems with very low churn rates) the lease mechanism has a valuable impact as shown in [21].

4.3.3 The Impact of Bootstrapping in SCAMP

In these simulations we want to evaluate the SCAMP behavior under the condition in which leaves occur only after the overlay has reached a size equal to a good fraction of the total number of node involved. For this reason we have considered that in the bootstrap phase the protocol builds a starting group of $n_1 = \frac{1}{4}n_{tot} = 40$.

Evaluating the join dynamics rate in $\Delta_b$. SCAMP is sensitive to different join dynamics rates (number of subscriptions per second) injected in the bootstrap interval to get quota $n_1$. The following plots (Fig. 4.8(a) and Fig. 4.8(b)) shows that the $n_1$ nodes are better connected if the join dynamics rate is at most equal to 2 joins per second. Note that the partial view size (in the growing phase to equal the node degree) does not increase even if the rate is lower than 2s$^{-1}$. Then, in the following we consider a bootstrap interval of 40s, i.e. a join dynamics rate of 1s$^{-1}$.

Evaluating Reliability of the Topology generated by SCAMP. In the plots of Fig.4.9 the effect of bootstrapping is evaluated in three different scenarios: (i) the whole group of nodes obtained after the bootstrap stably remains in the overlay for the entire
duration of the perturbation, (ii) only half of the group obtained after the bootstrap remains stably in the overlay for the perturbation duration and (iii) the whole group obtained after the bootstrap leaves the overlay during the perturbation. The worst case is in the third scenario. At first sight it could seem surprising that this scenario is even worse than the scenario with no bootstrapping (see Fig. 4.2(a)). The explanation behind this behavior lies in the fact that in the bootstrap interval SCAMP creates a cluster very well-connected of 40 nodes. This cluster remains poorly connected with other nodes that successively enter the group during the perturbation\(^6\). Thus, if nodes of this cluster leave (0% stable nodes), the graph at the end of the perturbation interval is poorly connected or may be directly partitioned. For the same reason, the reliability provided by SCAMP in the best case, showed in the first scenario, does not depend on the churn rate.

\(^6\)This is due to the fact that nodes joining the overlay during the perturbation may never get the forwarded subscriptions of the nodes forming the cluster, as the forwarding is already over. In this case the new nodes may never add in their partial views the identifiers of cluster nodes.
Figure 4.8: Measures taken at time $t_1$ after a join dynamics rate of $40/\Delta_b$.

![Graph a](image1.png)

![Graph b](image2.png)

Figure 4.9: SCAMP reliability with $\Delta_b > 0$ and a $\Delta_p$ with churn rate equal to $200/\Delta_p$.

![Graph a](image3.png)

![Graph b](image4.png)
Chapter 5

Related Work

In the following the state of the art concerning solutions for large scale dissemination and content-based searching applying in P2P systems is described. For each work will be highlighted under what assumption they perform well. Our work has also some relationship with the work done in group membership for group communications, then its the state of the art is briefly described and the position of our work in this context is pointed-out.

5.1 Group Membership for Group Communications

The group membership problem has been extensively studied and many implementations and specifications exist in the literature.

Traditionally, group membership supports group communication. Group communication is an useful abstraction in which processes are arranged in groups and communicate by sending message to a logical name identifying the group. To support group communication, group membership has the objective of maintaining a list of currently up and connected processes in the group. This list is called view. This list changes with new processes joining and other departing or failing. In this case a new view is installed at mutually connected processes in the group. Multicast services that deliver messages to the current view members complement the membership services. A group membership service may either be primary partition or partitionable. A primary-partition group membership service ensures that the processes agree on the sequence of installed views, i.e views are totally ordered. In partitionable group membership the set of views is partially ordered, i.e. multiple disjoint views may coexist concurrently.

Many systems have been developed that provide group membership services. The first of these systems is the Isis toolkit[9]. Others include the Highly Available System[14], Transis[4], Newtop[19], Totem[51], Horus[50], Spread[5], Relacs[40], Moshe[31]. Each one of these systems offers a group membership service with a different specification. In [13] thirty published systems offering a group membership service are surveyed. [13] classifies existing group membership services providing a comprehensive set of rigorous specifications. To reason about relations between the group membership problem and the connectivity
maintenance problem we will refer to these specifications.

Roughly speaking, the connectivity maintenance problem does not address any of the Safety properties of group membership (e.g. virtual synchrony, view ordering, etc.) as it does not offer any support to view-oriented communication (communication within views). The connectivity problem, well-suited in large scale systems, implicitly takes into account a need of locality allowing the maintenance of partial local views (neighbor variables) at each process. The locality principle allows concurrent installation of different views at each process, e.g. two joins triggering the concurrent installation of two different and disjoint views at two vertices. On the other hand, connectivity predicates on the transitive closure of connectivity relations defined by local views, that has to comprise all current members (overlay vertices). This concept is very similar to the concept of precise view in group membership: a view that comprises all processes that are currently up. For this reason, the connectivity maintenance problem has strong similarities with Liveness properties of group membership. To appreciate this relationship, let us detail the Liveness properties specified in [13]. These Liveness properties, namely Precise Membership and Membership Accuracy (see below) are conditional on the underlying network behavior. In particular, Precise Membership delivers a view that correctly reflects the network situation to all live processes comprised in the so-called Stable Component. Formally:

**Stable component.** A stable component $S$ is a set of processes that are eventually alive and connected to each other and for which all the channels to them from all other processes (that are not in the stable component) are down.

**Precise membership.** A membership services is precise if for every stable component $S$, there exists a view with the members of $S$ which is the last view of every process in $S$. Precise membership is guaranteed only in those runs in which the network eventually stabilizes. The network stabilizes if from some point onward (i) no process crash or recover, (ii) communication is symmetric and transitive and (iii) no changes occur in the network connectivity. As the authors state in [13] the presence of eventually perfect failure detectors is prerequisite for Liveness. In our model, a stable component is the set of stable processes when joins subside. To satisfy connectivity only one stable component is created. The last view can be considered as the global view defined by the transitive closure of connectivity relations defined by local views. As seen in Chapter 3, even assuring Eventual Connectivity in presence of failures is conditional on the stabilization of the overlay. The reason why in our implementation (see Section 3.2) eventually perfect failure detectors are not needed depends on the fact that this global view may comprise even faulty/out processes like in [31]. On the other hand, strengthening the connectivity specification to purge from the global view non-live vertices, the need of an eventually perfect failure detector arises.

**Membership Accuracy** (originally stated in [40]) states that if there exists a time after which two processes $i$ and $j$ are alive and the channel from $j$ to $i$ is up, then $i$ eventually installs a view that includes $j$ and every view that $i$ installs afterwards also includes $i$. The comparison between this property and Connectivity points out the different nature of the two problems: the concept of “channel” in an overlay is a logical concept defined by local view relationships. By definition of connectivity: there exists a channel, i.e. a connection between two processes, iff one process includes the other in its local view. For this reason
Membership Accuracy does not apply in our context.

5.2 Maintenance of Unstructured Overlays

Unstructured overlays have not a deterministic topology to maintain. The goal addressed by unstructured overlays is supporting efficiently large scale information dissemination and in some cases content-based searching (flooding-based)\cite{41} but there is no mapping between content (e.g. data) and location (e.g. node address) as in structured overlays. The goal of a connectivity maintenance protocol for unstructured overlays is building, in a scalable manner, overlay topologies showing good global properties as connectivity, low-diameter and constant-degree. Two main classes of connectivity protocols for unstructured overlays have emerged: (i) gossip-based connectivity protocols\cite{18,21,3} and (ii) non-gossip based connectivity protocols\cite{37,30,33,41}. These two classes of connectivity protocols have originally emerged for supporting two different information dissemination approaches, namely, gossip-based dissemination and overlay-based dissemination.

Gossip-based dissemination are being widely studied because it is easy to deploy, scalable and robust\cite{23}. In a gossip protocol each node, on receiving a message, forwards this message to a small set of nodes chosen uniformly at random from the entire group of nodes. The resulting “epidemic” yields to probabilistic guarantees of delivery to all nodes. Epidemic algorithms was first introduced in\cite{16}. Many gossip protocols, while are scalable in terms of the communication load imposed on each node, rely on a non-scalable membership algorithm since they need to maintain global membership knowledge in order to obtain a uniform sampling of the entire group\cite{10}. This has motivated work on distributing membership management\cite{18,21,3} in order to provide each node with a small local view of the membership at each node. In all these approaches the membership information spreads in an epidemic style\cite{22}, i.e. these protocols are gossip-based connectivity protocols. In particular, in Lpbcast\cite{18} nodes periodically gossip a set of subscriptions they heard about during the last period to a random subset of other nodes (this number is defined by the fan-out), chosen from their partial view. A node receiving such a gossip message updates its partial view by replacing a randomly chosen node-id with a newly received one, and gossips the node Id removed from its partial view. The authors prove through an analytical analysis that the protocol achieves a good randomization of partial views and then high probability of delivering messages in a few rounds. The analysis points-out how the size of the partial view has no impact on the event propagation. This analysis is conducted assuming that: the composition of the group does not vary during the run, the system is loosely synchronized (each node is update exactly once in each round), failures are stochastically independent, fan-out and partial views are fixed. Unfortunately, this independence between partial view size and reliability is not completely confirmed by their simulations, i.e. the views obtained in practice with Lpbcast appear to not to be completely uniform and independent. This result is also confirmed by Jelasity et al. in\cite{29}, through an extensive and valuable experimental analysis (not comprising\cite{21}) point out the inability
of gossip-based implementations to build local views representing uniform sampling of the entire overlay. This renders traditional theoretical approaches invalid, when the overlay is defined by a gossip-based connectivity protocol. Let us remark that Lpbcast considers the fan-out as fixed. In [32] the authors derive the fan-out required to achieve high reliability as a function of the system size. Scamp [21] (widely described in Chapter 4) exploits this result and adapts, as nodes join and leave the group, the partial view sizes automatically in proportion to the logarithm of the number of members in the group. This work has pointed-out the relation between the overlay topology maintained and the efficiency of the supported gossip-based dissemination, in particular the authors of Scamp make an explicit attempt to build a (static) random graph topology [48].

These works does not take into account the issue of churn explicitly. Even though Scamp considers the possibility of a changing system size, there is no attempt to adapt the behavior of the protocol to different churn rates. In particular, for a good behavior, Scamp relies in a uniform random choice of the contact upon joining. The indirection mechanism tries to implement this assumption but is not clear what is its behavior under churn. In the recent work of Allavena, Demers and Hopcroft [3] a new scalable gossip based protocol for local view maintenance facing the problem of churn has been proposed. The authors evaluate the time expected until the overlay partitions as function of the overlay size, the local view size and the churn rate. The churn rate is modelled as a constant number of nodes that departs and that join at each round. The local view size needs to be larger than the churn rate for the expected time until partitioning to be exponential in the square of the local view size. This result, theoretically proved, holds without requiring uniformly random views. By simulation study it is confirmed that increasing view size or decreasing the churn rate increases the time until the overlay partitions. The protocol proposed, however, has to know a-priori the overlay size and the churn rate to tune its parameters. Mechanism to dynamically adapt as churn rate and system size changes are not investigated.

Overlay-based dissemination has as main target the construction of a useful topology able to efficiently support information dissemination. In this approach the dissemination is not strictly gossip-based: a node does not choose at random from its view in each round but always communicate with its neighbors in the overlay. The tree topology, is one of the most widely used overlay topology for information dissemination [28, 11]. In this case, mechanism to recover from message losses must be provided as the tree will frequently become partitioned due to failures. This can inhibit scalability if failures are frequent. Recently, several approaches to the construction of topologies based on a randomized choice of neighbors [33, 41] have proposed. The construction and maintenance of these graphs is not gossip-based. In [33] the authors present a distributed algorithm for building random expander overlay topologies that are composed of $d$ Hamilton cycles. In particular, the constructed topologies are expanders with $O(\log d \times n)$ and $O(\log n)$ diameter with high probability. To achieve this goal a joining node has to search and locate a particular (random) position before adding the overlay. These type of constraints on the overlay access are criticized in [41]. The authors point out that an incoming node (like any node in the network) does not have global knowledge on the current topology, or even the identities of other nodes in the
current network, then is unfeasible to require an incoming node to connect to some random nodes in the hope of creating an expander network. To this aim the authors propose a local heuristics that leads to the formation of a connected overlay of constant degree and logarithmic diameter with high probability. The protocol relies on a central server that maintains a cache containing a constant number of nodes through simple replacement/joining rules (of constant overhead). The drawback is that the proposed solution requires a central server to guarantee connectivity that could represent a bottleneck with high join rates. In other words, churn is controlled by serializing joins. Moreover, the analytical model considers a Poisson model of arrivals and departures. Recently, Stutzbach and Rejaie in [49], characterizing churn in Gnutella Network [1] basing on a fine-grained monitoring of the entire population, have observed that the behavior of arrival and departure follows a power-law distribution, thus contradicting previous studies assuming a poisson distribution of arrivals and departures.

5.3 Maintenance of Structured Overlays

Structured overlays have stringent neighbor relationship to be maintained. Many of them [27, 47, 46, 52] support efficient data location and discovery basing on Distributed Hash Tables which provide a many-to-one mapping between data items and peers. The mapping is accomplished by organizing peers in some virtual coordinate space and hashing each data item to these virtual coordinates. Thus, each data item is identified by a key and nodes are organized into a structured graph that maps each key to a responsible node. The data or a pointer to the data is stored at the node responsible for its key. These constraints provide efficient support for exact-match queries; they enable discovery of a data item given its key in only $O(\log N)$ hops with only $O(\log N)$ graph neighbors per node.

Many different topologies have been proposed. In Chord [27] each node and data item is associated with a unique key in an uni-dimensional space, i.e. a ring. In Pastry [46] the nodes are arranged in $k$-regular graphs. In Bayeux [25] and Tapestry [52] the topology is based on prefix-based routing [42]. In CAN [47] the topology is a $d$-dimensional cartesian space. Other proposals concern skip graphs [7], de Bruijn graphs [30], butterfly topologies [37]. A graph-theoretic analysis of these different topologies is provided in [35]. As argued in Liben-Nowell et al. [15] all these approaches do not fully consider the evolutionary nature of P2P systems. This work points out what are the characteristics of a P2P system. Without formalizing a computational model they take into account all the sources of non-determinism afflicting P2P systems. In this respect, they first define a metric called half-life of the network, which essentially measures the time for replacement of half the nodes in the network by new arrivals. Then, they theoretically prove that to maintain connectivity there exists a lower-bound on the generated traffic. In particular, they prove that $\Omega(\log N)$ nodes has to be notified of new nodes in the network with a frequency defined by the half-life. Then, they propose a maintenance protocol that with high probability repairs a ring topology over the time. This protocol works under the following assumptions: the half-life is
known, the system size is known, arrivals and departures follow a Poisson model. Moreover, they assume perfect failure detection.

The work closest related to our implementation part in absence of failures is [34]. The authors present a maintenance protocol for ring topologies able to manage concurrently joins/leaves and in which leaves are handled actively. Through an assertional proof method they prove that the ring topology is eventually restored after membership changes subside. They also propose the maintenance of a ring-based redundant structure, namely a ranch topology. However, the model they assume considers a finite number of processes and they do not deal with cumulative failures. Moreover, as just discussed in Section 2.1.4 they rely on some assumptions restricting asynchrony.
Conclusions and Future Work

This work has provided a better and formal understanding of P2P distributed systems. Key contributions have been the analysis of a P2P distributed system in order to capture all possible forms of non-determinism it suffers from, and solutions to the overlay connectivity maintenance problem.

A computational model has been formally defined and the connectivity maintenance problem has been formally specified. Differently from traditional computational models for distributed systems, the proposed distributed system model incorporates the concepts of unbounded concurrency and infinite arrival model. This leads to cope with the so-called churn, i.e. (i) an unpredictable number of processes that simultaneously join/leave the system and (ii) an infinite number of processes that continuously join and leave the system.

The connectivity maintenance problem has been specified giving a hierarchy of four specifications. In particular, we have defined two classes of specifications: the strongest one requires connectivity at any time, the other requires that eventually connectivity will be assured. By presenting a set of protocols along with their correctness proof, it has been shown that in absence of failures, connectivity at any time can be assured. However, the level of asynchrony impacts on the memory required. Moreover, churn and asynchrony affect unfavorably the maintenance of a simple topology like a tree. It has been shown that the overlay topology is an undirected tree only when the overlay stabilizes. The implementation under (i) asynchrony, (ii) churn, and (iii) failures has been also investigated. In this case, it has been discussed, guaranteeing connectivity at any time turns in relying on meaningless assumptions. Then, the implementation of eventual connectivity has been taken into account. In this case guaranteeing connectivity is conditional on the stabilization of the overlay and the underlying network behavior. Through an experimental analysis which compares a probabilistic maintenance protocol and one of the deterministic proposed protocol, it has been pointed out a sharp trade-off between reliability of the generated overlay topology and its ability to scale under churn.

Let us remark that the work herein presented is not focused on solving a particular well-known problem under a well-defined model. On the contrary, this work pretends to open new research issues by defining a formal framework in which useful solutions should be designed. To this end, relying on the findings of this thesis, future work should focus on:

- identifying solutions for controlling non-determinism without limiting it, as much as possible, by assumptions. Specifically, investigating new ways for coping with the
churn – this Thesis has shown that churn has a disrupting effect on P2P systems and it has proposed an active approach for leaves:

- devising protocols with adaptive and local behavior – by relying on useful heuristics a node monitors system conditions and adapts consequentially;

- quantitative characterization of existing P2P systems (actual churn rates, failure distribution, etc.); and

- empirical validation of existing and newly proposed solutions.

Concluding, we can state that another building block towards understanding P2P systems has been placed, but still a lot of work remains to be done.
Bibliography


