TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MAGISTERARBEIT

Distributed Construction of a Fault-Tolerant Wireless Communication Topology for Networked Embedded Systems

oder “Implementing the Thallner-Algorithm”

Ausgeführt am
Institut für Technische Informatik
der Technischen Universität Wien

unter der Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Ulrich Schmid

durch

Heinrich Moser

Reisenbauerring 7/2/3
A-2351 Wiener Neudorf

16. November 2006

Abstract

This master's thesis presents a proven-correct implementation of a distributed topology construction algorithm based upon the Thallner topology construction method for creating a minimal Δ-node connected fault-tolerant overlay graph. The algorithm works in asynchronous fault-tolerant distributed systems augmented with failure detectors. A detailed proof shows that given a perfect propose module and a period of network stability, the unique minimal overlay graph is built. This thesis also contains a solvability analysis examining how the algorithm can be implemented in the presence of simple crash failures, in the crash-recovery model and in the presence of lossy links.[1]

Zusammenfassung

Diese Magisterarbeit präsentiert eine bewiesenermaßen korrekte Implementierung eines verteilten Topologiekonstruktionsalgorithmus basierend auf der Thallner-Methode zur Topologiekonstruktion, mit der ein minimaler Δ-knoten-verbundener fehlertoleranter Überlagerungsgraph erzeugt werden kann. Der Algorithmus funktioniert in asynchronen fehlertoleranten verteilten System mit Fehlerdetektoren. Ein detaillierter Beweis zeigt, dass, falls ein perfektes Propose Module zur Verfügung steht und das Netzwerk stabil genug bleibt, der minimale Überlagerungsgraph erzeugt wird. Diese Magisterarbeit enthält weiters eine Lösbarkeitsanalyse, die untersucht, wie der Algorithmus bei einfachen Crash-Fehlern, im Crash-Recovery Modell und bei Nachrichtenverlust implementiert werden kann.

[1] This work has been supported by the EC FP6 Integrated Project ASSERT IST-004033, `http://www.assert-online.org`.

Contents

1 Introduction

Thallner et. al. [TS04] published a distributed algorithm for constructing a sparse overlay network that has fixed node-degree and facilitates efficient fault-tolerant multi-hop communication in large-scale distributed systems. This paper presents a full pseudo-code implementation of the algorithm for asynchronous systems with failure detectors and node crashes and proves its correctness.

1.1 Wireless Overlay Networks

Although the algorithm can be used for creating communication topologies outside the field of wireless communication, the specific requirements imposed by this technology make this approach particularily useful in this area of networking.

In computer science, a network is a collection of *nodes* (or *processors*—the terms will be used interchangeably), connected through *links*, similar to the mathematical concept of a *graph* consisting of *vertices* connected by *edges*. Usually, the nodes represent some sort of processing units (e.g. computer devices) which exchange some kind of data over the links either by *direct communication*, i.e. by directly transmitting a message from the sender to the receiver, or by *multi-hop communication*, i.e. by passing the message on from node to node until it reaches the receiver. *Wireless* network do not have a wired infrastructure providing these links; thus, wireless devices have to build and maintain *wireless links*. Two problems arise out of this:

- Wireless devices have a maximum transmission range.
- The *cost* for sending messages or maintaining links between two nodes at distance d is proportional to d^k, $k \geq 2$. Thus, it is cheaper to use two short links rather than one large link. (See Figure 1.1 for an example.)

The first issue implies that in many cases multi-hop communication is a neccessity; the second one claims that multi-hop communication is more energy efficient than direct message transmission. Thus, it makes sense to create a suitable *network topology*, i.e. a set of well-chosen links over which the nodes communicate. We call the cost-weighted graph induced by direct connections the *underlying communication graph* and the created network topology the *overlay graph*.

Ad-hoc networks are a special class of wireless networks which are constructed on demand and have to cope with problems such as moving nodes, new nodes and crashing/removed nodes. Thus, the topology for ad-hoc networks must be able to adapt to this changing environment.

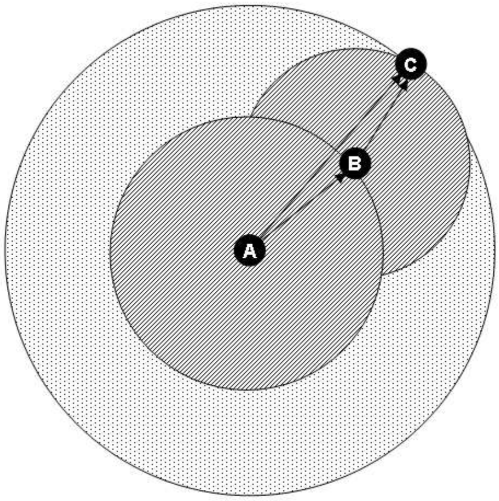

Figure 1.1: *Example comparing transmission cost for direct vs. multi-hop communication in the ideal case $k = 2$.*

1.2 Principles of Computer Science

This section includes introductory material and definitions in fundamental areas of computer science, which are necessary to understand this master's thesis.

1.2.1 Automata Theory

In theory, a processor is seen as a *state machine*. A (deterministic) state machine (or *automaton*) is an abstract machine with a set of *states* (Q), a set of *symbols* (Σ), a transition function ($\delta : Q \times \Sigma \Rightarrow Q$), an initial state ($S_0$) and a set of end states (F) [wp:04]. As an example, consider the following simple algorithm running on a processor:

```
var i := 0
do
  read m
  if m = A
    i := 0
  if m = B
    i := 1
loop until m = C
```

The state machine would look like this:

- *States*: $\begin{array}{ll} q_0\text{: } i=0\text{, running} & q_1\text{: } i=1\text{, running} \\ q_2\text{: } i=0\text{, terminated} & q_3\text{: } i=1\text{, terminated} \end{array}$
- *Symbols*: A, B, C
- *Transition function*: $\begin{array}{llll} (q_0, A, q_0) & (q_1, A, q_0) & (q_2, A, q_2) & (q_3, A, q_3) \\ (q_0, B, q_1) & (q_1, B, q_1) & (q_2, B, q_2) & (q_3, B, q_3) \\ (q_0, C, q_2) & (q_1, C, q_3) & (q_2, C, q_2) & (q_3, C, q_3) \end{array}$
- *Initial state*: q_0
- *End states*: q_2, q_3

Figure 1.2 contains a graphical representation of the transition function.

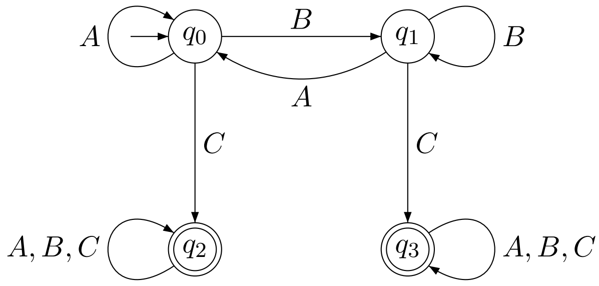

Figure 1.2: *Example state machine.*

Basically, a pseudo code algorithm can be translated into the state machine model by identifying states (with a state representing the current variable assignments and the current code position, i.e. the instruction pointer) and the corresponding transition functions (based on the control flow of the program). As we are analyzing deterministic state machines, states with trivial transitions (only one possible successor) can be left out. This is the reason why the above example does not contain a separate state for every line of code.

1.2.2 Distributed Algorithms

The challenge of a distributed algorithm is to solve a global problem (e.g. the construction of the above mentioned overlay graph) in a distributed fashion, i.e. there is no central, coodinating computing unit which solves the problem and informs the other nodes or coordinates the computation. The same code[1] is executed on every processor and the system should continue to work if arbitrary processors crash.

[1] Strictly speaking, it is not the same code, as each processor has a different ID it can use in the code.

Formally, the system is modelled as follows (compare to [AW04]): A system consists of a set of n processors $p_0, \ldots, p_{n-1}$. Each processor is a state machine as defined above. Between each two processors there can be a link. The processors can use the link to exchange messages. For each link, each node on the link has an in-buffer and an out-buffer variable. When a message send command is issued on one processor, the message is put in the processor's out-buffer for this link. When the message m is delivered from a processor p_i to processor p_j during a *delivery event* $del(i, j, m)$, the message is moved from the out-buffer of p_i to the in-buffer of p_j. During the next *computation event* on p_j, the message will be removed from the in-buffer and processed.

A *configuration* is a vector $(q_0, \ldots, q_{n-1})$ containing the states of all processors $p_0, \ldots, p_{n-1}$. Note that a delivery event changes the state of two processors, whereas a computation event changes the state of exactly one processor. An *initial configuration* is a configuration where each processor is in an initial state.

An example for a distributed algorithm would be the example code of the previous section with *read m* replaced by *wait for some message m*.

1.2.3 Graph Theory

As mentioned above, a graph is a set of vertices (or *nodes*), some pairs of which being connected by edges. Figure 1.2 is also an example for a directed graph. From now on, we will look at undirected graphs without loops. Formally, such a graph is defined as a set of vertices V and a set of edges $E \subseteq V \times V$ with $\forall x : (x, x) \notin E$. The *degree* of a node is the number of edges connected to it. A graph is

- *k-regular* if every vertex has degree k.
- *connected* if, for every pair of vertices, there is a path (i.e. a sequence of edges) connecting these vertices.
- *k-connected* if, even after removing $k - 1$ arbitrary vertices, the graph is still connected, whereas the removal of k vertices disconnects the graph. This implies that between every pair of vertices there are k vertex-disjoint paths, by Menger's theorem.
- *fully connected* if there is an edge between every pair of vertices.

1.3 Construction Method[2]

1.3.1 Idea

The Thallner construction method is based upon a clustering scheme introduced in [Tha04b], which recursively forms groups consisting of Δ nodes that

[2] This section is joint work with Bernd Thallner and reviews results from [TS04].

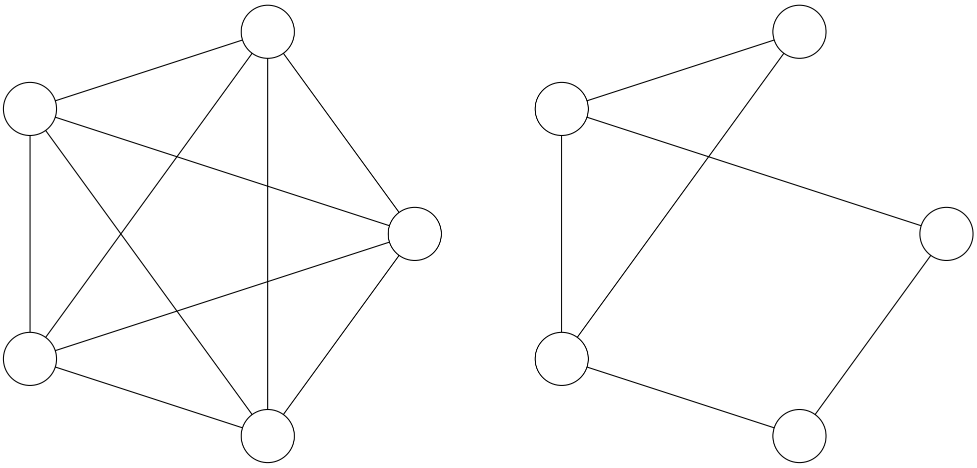

Figure 1.3: *Example of a fully-connected and a 2-connected graph.*

are treated like single nodes subsequently. In [TS04], this scheme was extended to a fully distributed algorithm for constructing and continuously maintaining a Δ-regular and Δ-connected overlay graph for fault-tolerant multi-hop communication in large-scale wireless networks. See [TS04] for a in-depth analysis of the method.

The algorithm consists of two reasonably independent parts, which allow to adapt this scheme to very different wireless networks:

1. The generic construction algorithm (presented in Section 3), which builds up and continuously maintains the Δ-regular and Δ-connected overlay graph. It does so by processing proposals for links to be added to the overlay graph supplied by the specific propose module (see next item).

2. The propose module (cp. Definition 4 and Section 7 in [TS04]), which tries to find minimal-weight links to be added to the overlay graph. The propose module is network-specific and allows to trade construction complexity for minimality of the weight-sum of the overlay graph (and hence overall energy efficiency, for example).

The analysis presented in this thesis will reveal that the algorithm always converges to the unique minimal topology, for any reasonable propose module. The eventually constructed overlay graph is Δ-connected, which is optimal, and thus ensures that Δ node-disjoint paths exist between any pair of nodes. It has low total weight and inherently provides failure-locality as well: Even excessively many failures in some part of the system do not impair fault-tolerant communication in other parts. As a by-product, the algorithm produces a hierarchy of clusters represented by a Δ-ary tree that reflects the node "density". This property can be used in higher level services, like data aggregation in sensor networks, routing, naming, as well as geo- and multicasting.

Whereas the convergence of our algorithm is independent of the choice of the propose module, (worst case) performance is obviously not: The worst case message complexity for constructing an overlay graph with n nodes ranges from $O(n)$ to $O(n^{\Delta+1})$, depending upon the propose module used. Similarly, the worst case time complexity ranges between $O(n)$ and $O(n^2)$. On the other hand, simulation results show good (linear) average complexity and small spanning factors for most propose modules.

1.3.2 Definitions

Consider a simple undirected weighted graph $G = (\Pi, \Lambda)$ consisting of a set of n *nodes* $\Pi = \{1, ..., n\}$ and a set of weighted *edges* $\Lambda \subset \Pi \times \Pi \times \mathbb{R}$. The network is modeled as a *communication graph* G and contains the set of potential edges. It is is assumed to be fully connected, in the sense that $w < \infty$ for any edge $(x, y, w) \in \Lambda$.[3] Note carefully that this assumption does not mean that any node actually communicates with every other node, but only that it could communicate with every (reasonable) peer, if necessary. Both the set of alive nodes and the weight of the edges in the communication graph G may be time-variant.

The algorithm constructs a low weight *overlay graph* G' that is Δ-regular and Δ-connected, for some given Δ. Recall from Section 1.2.3 that a graph is *κ-connected* (also referred to as κ-node-connected or κ-vertex-connected) if the removal of any subset of $\kappa - 1$ nodes leaves the graph connected while there exists a subset of κ nodes whose removal disconnects the graph. A graph is *regular* of degree r if all nodes have degree r. In order to easily distinguish the communication graph G and the overlay graph G', the edges of the latter will be called *connections*.

In order to avoid the special top-level group of the overlay graph, employed in [Tha04b], [TS04] introduces *gateway nodes* and assumes that a small number of them ($n'' \geq 2\Delta - 2$ are sufficient, cf. Theorem 1) are present in the network. In addition to the wireless communication links to/from regular nodes, which have to be set up by the algorithm, all gateway nodes are assumed to be fully interconnected with all other gateway nodes via a dedicated backbone network. The set of nodes Π hence consists of n' *regular nodes* Π' and n'' *gateway nodes* Π'' with $n = n' + n''$ and $\Pi = \Pi' \cup \Pi''$. In order to ensure that gateway nodes are only used after all regular nodes have been exhausted, it suffices to assume that the edge weights between regular and gateway resp. gateway and gateway nodes are chosen according to $\forall x \in \Pi', y \in \Pi'' \Rightarrow (x, y, \Delta^2 K) \in \Lambda$ resp. $\forall x \in \Pi'', y \in \Pi'' \Rightarrow (x, y, 2\Delta^2 K) \in \Lambda$, where K is the maximum edge weight between regular nodes. The algorithm will then construct a low weight overlay graph $G' = (\Pi' \cup B, C)$ with $B \subseteq \Pi''$ and $C \subseteq \Lambda$. Note that the regular nodes in

[3] This work does not consider the extension presented in Section 6 in [TS04], which would allow the fully connected assumption to be dropped.

G' have degree Δ; gateway nodes may have degree $\Delta - 1$ due to the additional backbone interconnection.

1.3.3 Clustering Scheme

This section provides an overview of the clustering scheme, which induces an overlay graph G' with the desired properties. The idea is to build groups of nodes that appear like single nodes, such that they can be treated like those subsequently.

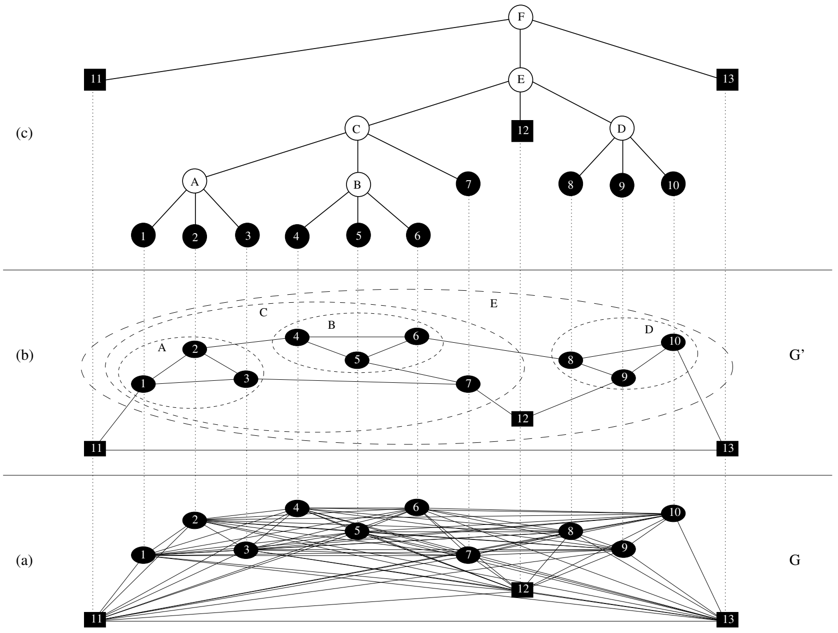

Figure 1.4: *Communication Graph (a), Network Graph (b) and Topology (c)*

Figure 1.4, taken from [TS04], shows an example, where 1.4a is the fully connected communication graph G, 1.4b depicts the constructed overlay graph G' for $\Delta = 3$, and 1.4c provides the tree representation corresponding to the constructed topology. From 1.4b it is apparent that the regular nodes $(1, 2, 3)$, $(4, 5, 6)$ and $(8, 9, 10)$ are combined into groups with id A, B, and D, respectively. Such a group is formed if all members agree upon the fact that the sum of the weights of their internal connections (e.g. $4 - 5$, $4 - 6$, $5 - 6$) is minimal over all alternative group constructions. Each of the Δ members of a group is connected to all of the $\Delta - 1$ other members (internal connections) and has exactly one connection left (external connection). Since there are Δ members in a group, any group has Δ external connections left, which are available in higher level groups. From the point of view of higher-level groups, groups hence look like nodes.

For example, group C again consists of three members: A single node 7 and two groups A and B, which are connected via their external connections. Again, all members of C agree upon minimality of the sum of their internal connections' weights. Groups E and F finish the topology and include gateway nodes ($11, 12$ and 13), which may have degree $\Delta - 1$ due to the additional backbone connectivity. Figure 1.4c reveals that the resulting group structure is a Δ-ary tree. The edges of the tree represent the membership relation among the groups.

The topology can be formally described as follows: A *group* consists of an identifier g_i and a set of $members(g_i)$. The set of all group identifiers is $\mathbb{G}$. The set of $members(g_i)$ consists of exactly Δ nodes and groups: $members(g_i) \subseteq (\mathbb{G} \cup \Pi)$, $|members(g_i)| = \Delta$. A node or group can only be member of a single group[4] $\forall g_a, g_b \in \mathbb{G}, g_a \neq g_b : x \in members(g_a) \wedge y \in members(g_b) \Rightarrow x \neq y$. For every $g_i \in \mathbb{G}$ the nodes of a group are defined as $nodes(g_i) = \bigcup_{k=0}^{\infty} members^k(g_i) \cap \Pi$ where

- $members^0(g_i) = members(g_i)$ and
- $members^j(g_i) = \bigcup_{k \in members^{j-1}(g_i) \wedge k \in \mathbb{G}} members(k)$.

For every node $p \in \Pi$ let $nodes(p) = \{p\}$.

A connection $(p, q, w) \in C$ from node p to node q with weight w is a *group g_i internal connection* if $p \in nodes(g_a)$ and $q \in nodes(g_b)$ with $g_a, g_b \in members(g_i)$ and $g_a \neq g_b$. If there is a connection $(p, q, w) \in C$ between node $p \in nodes(g_a)$ and node $q \in nodes(g_b)$ we call the groups g_a, g_b connected. The members of a group are fully connected among them: $\forall g_a, g_b \in members(g_i), g_a \neq g_b \Rightarrow \exists p \in nodes(g_a), q \in nodes(g_b), (p, q, w) \in C$.

Hence, every group member has $\Delta - 1$ connections to other group members and therefore exactly one connection left. Since there are Δ members in a group, the group has—like a node—Δ connections left. We call the Δ nodes of a group g_i with one connection left the *terminal nodes* $T_{g_i} \subseteq \Pi$ of a group. By convention, we define that $T_p = p$ for a single node $p \in \Pi$.

Definition 1. *The* weight of a group $\omega(g_i)$ *is a triple* $(A_i, members(g_i),$*internal connections of group* g_i*), where* A_i *is the maximum of the sum of all group* g_i *internal connection weights and the maximum of all group members' weights plus an arbitrary small constant* ϵ*, formally:* $A_i = max(\sum internal\ connection\ weights, max(members'\ group\ weights) + \epsilon)$. *A group* g_i *has smaller weight than* g_j*, formally* $\omega(g_i) < \omega(g_j)$*, if* A_i *is smaller than* A_j *or, if equal,* $members(g_i) < members(g_j)$ *in lexical order or, if equal, the internal connections of group* g_i *are smaller than the internal connections of group* g_j *in lexical order.*

Note that this definition implies that the weight of a parent group is always higher than the weight of any of its members. This property is required for

[4] Note that the member function is not transitive.

ensuring that the minimum admissible overlay graph introduced in Definition 3 is well defined, and that the distributed algorithm converges.

Definition 2. *An overlay graph G' is called* admissible *if its corresponding topology has a single* root group g_{root} *where all terminal nodes are gateway nodes:* $T_{g_{root}} \subseteq \Pi''$.

Recall that by convention, it is assumed that all gateway nodes are fully connected (primarily via backbone connectivity).

The following Theorems (proven in [TS04]) show that no more than $2\Delta - 2$ gateway nodes are necessary to construct an admissible overlay graph and that there exists a unique *minimal* admissible overlay graph (which is not necessarily the global minimum-weight overlay graph). Informally, the minimum criterion for choosing group members requires that the selected group has minimal total weight of all internal connections, and that all members agree upon this fact. It may hence be the case that there is a lower-weight choice for some members, but no one that is better for all members of any alternative group.

Theorem 1. *For every graph G with $n' \geq 1$ and $n'' \geq 2\Delta - 2$, there exists an admissible overlay graph G'.*

Definition 3. *An admissible overlay graph G' is* minimal *if, for all members $x \in members(g_i)$ of any group $g_i \in \mathbb{G}$ in the corresponding topology tree, no other group g' can be built with $x \in members(g')$, $\omega(g') < \omega(g_i)$ and $\forall y \in members(g') : \omega(g') < \omega(g_y)$, where g_y is the (unique) group in the topology tree with $y \in members(g_y)$.*

Theorem 2. *For every graph G with $n' \geq 1$ and $n'' \geq 2\Delta - 2$, there is exactly one minimal admissible overlay graph G'.*

The following theorems establish some other important properties of these overlay graphs.

Theorem 3. *In every admissible overlay graph G' the node degree is bounded by Δ and all regular nodes have degree Δ.*

Theorem 4. *Each admissible graph G' with $n' \geq 1$ and $n'' \geq 2\Delta - 2$ has $\lfloor \frac{n-1}{\Delta-1} \rfloor$ groups.*

Theorem 5. *Each admissible graph G' with $n'' \geq 2\Delta - 2$ is Δ-connected.*

The above results reveal several general advantages of this approach. First of all, connection weights may be arbitrary; in particular, they need not to satisfy the triangle inequality. Moreover, by adding additional constraints to Definition 3, overlay graphs with specific additional properties can be built.

If the weights in the communication graph reflect physical distance, the topology construction scheme clusters nodes according to their spatial density. Nodes

```
if a new proposal is provided by a propose module            /* PROPOSE_GROUP */
  broadcast proposal to all participants
  if all particpants agree that the proposal is better than their current parent
    all participants join the proposed new group

periodically                 /* CHECK_GROUP */
  check for group consistency
  if group is consistent
    recalculate group weight
  else
    all participants leave the group
```

Figure 1.5: *Basic Algorithm*

that are close to each other will be near the leafs of the topology tree, which leads to a nice failure-locality property of our overlay graph: Catastrophic failures in a spatially localized area will affect communication only in the immediate neighborhood.

In fact, failures that hit some part of the tree do not severely—if at all—impair fault-tolerant communication in other parts: The proof of Theorem 5 reveals that failures that completely wipe out all members of some group g do not impair Δ-connectivity of the remaining tree that results from purging the subtree rooted at g. Moreover, all nodes within some intact subtree rooted at some member (or sub-member) of g are still $\Delta - 1$-connected with each other, since only the single external connection routed via g is cut by the failures in g.

1.3.4 Basic Algorithm

In the basic[5] algorithm (Figure 1.5) every existing group (that is, its terminal nodes) concurrently searches for the minimal-weight next-level group to join. For this purpose every node repeatedly generates proposals P, consisting of the group members, the group weight, the group internal connections and the group's terminal nodes, which are sent to (the terminal nodes of) the proposed group members for confirming minimality. Generating a new proposal is typically triggered periodically, to facilitate adaption to changed connection weights, or upon detection of a node crash or join. For convergence, the algorithm requires that, in infinite runs, this group check is initiated infinitely often.

To simplify description and analysis, the functionality of generating proposals is encapsulated in a *propose module* that must satisfy the following specification:

Definition 4. *A* propose module *generates proposals for groups consisting of* Δ *members,* Δ *terminal nodes, group internal connections and the corresponding group weight. A propose module is* perfect *if it eventually generates proposals corresponding to groups in the final (unique) minimal overlay graph* G'*.*

[5] See Section 6 in [TS04] for an extension of the construction algorithm, which keeps topology changes caused e.g. by a node failure in the vicinity of the failed node. This extension is particularly beneficial in very dynamic environments.

For a more detailed analysis of propose modules see [Tha04a] or Section 7 in [TS04]. Section 3, the main part of this work, deals with a proven correct pseudo implementation of this algorithm in asynchronous systems where non-blocking atomic commitment is possible.

1.4 Related Work

Study of other fault-tolerant topology control algorithms is outside the scope of this thesis, see Section 8 in [TS04] and the references mentioned there for an overview of different solutions.

So far the problem of implementing the "Thallner method" for distributed construction of overlay networks has not been addressed except for two still unpublished papers by Bernd Thallner, Ulrich Schmid and myself [TS04,MT04], which form the basis of this thesis.

1.5 Roadmap

This thesis is organized as follows: Chapter 2 describes the atomic commitment problem, the reasons for using it in the topology construction algorithm, and some extensions to the protocol. Chapter 3 contains the main part of this work: A pseudo code implementation of the distributed topology construction method. Chapter 4 proves the correctness of the algorithm. Chapter 5 analyzes the requirements for the system model in terms of synchrony assumptions such as failure detectors. These results are then extended to more challenging classes of systems such as the crash-recovery model and networks where messages can be lost. In Chapter 6 the results are summarized and pointers to further work are given.

2 Atomic Commitment

2.1 Introduction

As information about the system is not on a central node but distributed among different processors, both the group creation (i.e. proposal decision) phase and the periodic group checking must be coordinated between those processors and thus involve an underlying distributed agreement problem, whose solution must satisfy the following conditions:

PROPOSE_GROUP

- *Termination*: Eventually, every correct participant must decide on either COMMIT (join proposed group) or ABORT (do not join proposed group).
- *Uniform Agreement*: All participants must make the same decision.
- *Validity*: If the common decision is COMMIT, all participants must want to join the proposed group (because it has lower weight than the current group).
- *Non-triviality*: If no participant crashed and all participants want to join the proposed group, the common decision must be COMMIT.

CHECK_GROUP

- *Termination*: Eventually, every correct participant must decide on either COMMIT (recalculate weight of existing group) or ABORT (leave existing group).
- *Uniform Agreement*: All participants must make the same decision.
- *Validity*: If the common decision is COMMIT, all participants must have consistent information about the existing group.
- *Non-triviality*: If no participant crashed and all participants have consistent information about the existing group, the common decision must be COMMIT.

These conditions correspond to a well-known agreement problem called *atomic commitment*. The problem originated in the area of distributed databases where it is necessary to make changes permanent (COMMIT) only if this is possible on all participating databases. As we want the above properties to hold even when nodes crash during the negotiation process, we need a special variant called *non-blocking atomic commitment.* A comprehensive discussion of this topic and different algorithms solving this problem can be found in [BT93].

Note carefully, however, that convergence of the algorithm (i.e. eventual creation of the overlay graph) only requires the non-triviality condition to hold *eventually.* Due to the use of a perfect propose module, the formal correctness proof can in fact tolerate an arbitrary amount of incorrectly destroyed or incorrectly not-built groups. This corresponds to the *non-blocking weak atomic commitment*, which is weaker (i.e. easier to solve) than traditional non-blocking atomic commitment [Gue95]. Subsequently, we will use the term *atomic commitment* and the abbreviation *NBAC* to refer to this weak variant of the problem.

2.2 Structure

Throughout this work, the following terminology is used: On the initiator of some proposal, *initiate atomic commit* with parameters *data* and *participants* is called, which is usually implemented by reliably multicasting (*data*, *participants*) to all participants [Ray97]:

```
procedure initiate atomic commit ($data, participants$)
    multicast ($data, participants$) to $participants$
```

"multicast" denotes an implementation of a reliable multicast protocol, i.e. a protocol ensuring that the message is either sent to all participants or to none (if the node crashes before multicasting). See [Ray97] for a formal definition of this primitive. Further information about this problem can be found in [HT93].

After an atomic commitment sequence has been initiated on the participants, the protocol must

1. query the node for its vote (VOTE_COMMIT or VOTE_ABORT) and
2. inform the node about the global decision (COMMIT or ABORT).

[Ray97] suggests the following basic structure for a NBAC protocol (adapted to our terminology):

```
upon delivery of ($data, participants$)
    nbac($data, participants$)

procedure nbac($data, participants$)
    var $vote \leftarrow$ vote($data$)
    var $result \leftarrow \bot$
    send $vote$ to all $participants$
    wait for ($a$) (delivery of a vote VOTE_ABORT from a participant)
        or ($b$) (one of the participants is suspected to be faulty)
        or ($c$) (delivery of a vote VOTE_COMMIT from all participants)
    case
        ($a$), ($b$): $result \leftarrow$ Unif_Cons(ABORT)
        ($c$): $result \leftarrow$ Unif_Cons(COMMIT)
    decision($result, data$)
```

We provide two functions to parameterize the NBAC protocol: *vote*(*data*) is called to locally decide on VOTE_COMMIT or VOTE_ABORT, whereas *decision*(*result*, *data*) is executed on every participating node after the atomic commitment algorithm reached a common decision. For example, when a new proposal arrives, *vote*(*proposal*) would return VOTE_COMMIT if the node or group wants to join the proposed new group. Afterwards, *decision*(*result*, *proposal*) would be called after the NBAC protocol has terminated to give the node the opportunity to join the new group if *result* = COMMIT.

Unif_Cons(*proposed_value*) refers to a sub-protocol solving uniform consensus, a widely studied fundamental problem in distributed computing [PSL80, Ray97]. Informally speaking, uniform consensus guarantees that all participants agree on a common value, which must have been proposed by at least one participant. Note that uniform consensus behaves differently than NBAC in that *any* proposed value might be selected, whereas NBAC has strict rules on when the decision must be COMMIT and when it has to be ABORT.

Recall the following properties defined for NBAC in [Ray97]:

- *Termination*: Every correct participant eventually decides.
- *Integrity*: A participant decides at most once.
- *Uniform Agreement*: No two participants decide differently.
- *Validity*: If a participant decides COMMIT then all participants have voted VOTE_COMMIT.
- *Non-Triviality*: If all participants vote VOTE_COMMIT and there is no failure suspicion then the outcome decision is COMMIT.

2.3 Extensions

2.3.1 Commit Data

The first of our extensions (Figure 2.1) piggybacks additional information (called *commit_data*) on VOTE_COMMIT votes. This information is used to recalculate the group weight after CHECK_GROUPs.

Note that *global_commit_data* is only guaranteed to be consistent if the global decision is COMMIT and no node was suspected. Otherwise, one participant might not have received the VOTE_COMMIT votes of all participants. Formally, the following property is added to the NBAC protocol:

- *Commit Data Validity*: If a participant decides COMMIT and no other participant was suspected, its output includes all data provided by the participants when voting.

It is obvious to see that the original properties are not affected by the change.

```
procedure nbac(data, participants)        /* commit_data extension */
    var vote←vote(data)        /* returns VOTE or array (VOTE, local_commit_data) */
    var local_commit_data←⊥
    if vote is array
        local_commit_data←vote[1]
        vote←vote[0]
    var result←⊥
    send (vote, local_commit_data) to all participants
    wait for (a) (delivery of a vote VOTE_ABORT from a participant)
        or (b) (one of the participants is suspected to be faulty)
        or (c) (delivery of a vote VOTE_COMMIT from all participants)
    case
        (a), (b): result←Unif_Cons(ABORT)
        (c): result←Unif_Cons(COMMIT)
    var global_commit_data←array of commit data received from all participants
    decision(result, data, global_commit_data)
```

Figure 2.1: *Commit Data extension*

2.3.2 Finalizing NBAC

The next extension deals with the fact that in one particular case, it is necessary to notify the participants that the *decision*() procedure has been executed on every correct participant: After some groups or nodes have decided to join a newly proposed group it is necessary to postpone the periodic group checking (see Figure 1.5) until the group information have been updated on all participating nodes. Otherwise, the periodic group checking protocol could get the false impression that the group is inconsistent and destroy it, thereby preventing convergence of the algorithm. We can solve this problem by running two NBAC sequences in parallel. The following code calls the user-defined functions *vote*(), *decision*() and *finalize*(), where *finalize*(*result*, $finalize\_result$, *data*) is run after *decision*() has been executed on all participating nodes (if the result has been COMMIT). Two NBAC procedures *nbac_n*() parameterizable by *vote_n*() and *decision_n*() ($n \in \{1, 2\}$) are used as sub-protocols.

Formally, the following properties are added:

- *Local Finalization*: Eventually, *finalize*() will be called on each correct participant. This happens after *decision*() has terminated on this participant.
- *Finalization Agreement*: No two participants decide on different $finalize_result$s.
- *Finalization Validity*: If a participant decides on $finalize\_result$ = *result* = COMMIT, *decision*() has terminated on every participant.
- *Finalization Non-Triviality*: If there is no failure suspicion then $finalize\_result$ = COMMIT.

The correctness can be shown easily:

```
var $r \leftarrow \perp$
var $nbac\_id \leftarrow$ new unique id (e.g. sequence number)

upon delivery of $(data, participants)$        /* finalizing extension */
    co−begin            /* concurrent execution */
        nbac_1$(data, participants)$
        nbac_2$(data, participants)$

function vote_1$(data)$
    return vote$(data)$

function vote_2$(data)$
    wait until unblocked
    return VOTE_COMMIT

procedure decision_1$(result, data)$
    decision$(result, data, nbac\_id)$
    $r \leftarrow result$
    unblock vote_2

procedure decision_2$(result, data)$
    var $finalize\_result \leftarrow result$
    finalize $(r, finalize\_result, data, nbac\_id)$
```

Figure 2.2: *Finalizing extension*

- *Termination*: Follows from termination of nbac_1 and nbac_2.
- *Integrity*: Follows from integrity of nbac_1 (i.e. *decision_1*() is only called once) and the fact that *decision*() is only called within *decision_1*().
- *Uniform Agreement*: Follows from uniform agreement of nbac_1.
- *Validity*: Follows from validity of nbac_1.
- *Non-Triviality*: Follows from non-triviality of nbac_1.
- *Local Finalization*: *decision_2* can only be called after *vote_2* has been unblocked. *vote_2* is only unblocked after *decision*() has terminated.
- *Finalization Agreement*: Follows from uniform agreement of nbac_2.
- *Finalization Validitiy*: Assume by contradiction that some p_i decides on $finalize\_result$ = COMMIT although *decision*() on p_j has not terminated yet. If $finalize\_result$ = COMMIT on any node, every participant voted VOTE_COMMIT for nbac_2. Thus, *vote_2*() has been unblocked on p_j. However, *vote_2*() is only unblocked in *decision_1*() after *decision*() has terminated, which leads to the required contradiction.
- *Finalization Non-Triviality*: Follows from non-triviality of nbac_2.

The unique id is used to allow algorithms to match *finalize*() calls to their corresponding *decision*() calls if multiple NBAC sequences run at the same time.

See Section 5 for additional constraints (failure detectors, reliable links, etc.) imposed on the system by the requirement for a non-blocking weak atomic commitment protocol.

It should be implicitly clear from context which NBAC variant is used in the following code.[1] The terms NBAC and "atomic commitment" are used interchangably throughout this work, referring to the appropriate extension of the non-blocking weak atomic commitment protocol.

[1] The *commit data* extension will be used for CHECK_GROUP, whereas PROPOSE_GROUP will use the *finalizing* extension.

3 Construction Algorithm

This chapter deals with a proven-correct implementation for solving the problem of distributed construction of the Thallner overlay network according to [MT04].

3.1 Problem Definition

Basically, constructing the overlay graph is a graph theory problem. The input consists of a fully-connected weighted graph G, i.e. a set of nodes V, a set of edges $E = \{x \in P(V) : |x| = 2\}$ and a weight function $\omega_E : E \to \mathbb{R}$. The output is the unique overlay graph G', i.e. a set of nodes $V' = V$ and a set of edges $E' \subseteq E$, and information about the group membership of the nodes. Globally, this problem is easily solvable, as shown in Theorem 6 in [TS04]. When information is distributed among different nodes, the problem is more difficult. Intuitively, the proof of Theorem 6 in [TS04] cannot be used because in an asynchronous system consistency of information can only be guaranteed *eventually.* See the convergence proof in Section 4.3 for more detailed information.

To solve the problem in a distributed way, the problem must be defined accordingly. A priori, the global meaning of "existence of a group" is not defined. There are only local data structures on different nodes (i.e. the state of the nodes) and one node might think that some group A exists while another one already knows that it has been destroyed for some reason. Or, worse, some propose module might think that some group A exists, although, in the meantime, group A has been destroyed and another group with the same id has been created. Another problem is the dissemination of weight information: This newly created group with id A might have different weight than the "old" group with the same id, because its internal structure is different. (Recall that the id of a group depends only on the ids of the group's terminal nodes). Thus, wrong decisions can be made on nodes supplied with information that has not been updated yet.

Thus, the problem is redefined to be solved individually on each node as follows: The input consists of communication primitives, allowing the node to send messages to every other node, and a perfect propose module (see Definition 4). As output, the algorithm produces a set of overlay edges ("connections") with this node as an endpoint and some local membership information, such as which group this node belongs to and where to find information about higher-level groups. Note that the algorithm does not need to satisfy some *termination* property, it must only provide *convergence*, i.e. if the underlying graph stays stable long enough, the data structures containing the membership and connection information of the overlay graph must eventually be consistent and stable until new nodes are added, old nodes crash or edge weights change.

3.2 System Model

The presented algorithm requires the following:

- a perfect propose module on each node (see Definition 4),
- a non-blocking weak atomic commitment service (see Section 2.1),
- a fully-connected asynchronous system (see Sections 1.2.2 and 1.2.3),
- two primitives *make connection* and *cancel connection* that create and remove connections in the overlay graph,
- a *check group* signal triggering group consistency checking. This signal must occur infinitely often in infinite runs of the algorithm.

The algorithm then guarantees that:

- A consistent, Δ-connected overlay graph corresponding to Theorem 2 is produced.
- The overlay graph is eventually stable and consistent, i.e. if there are no changes of the weight of the connections and no nodes are added or removed, eventually there will be no more changes to the structure of the overlay graph and all neighbors know each other.
- The algorithm tolerates and adapts to node failures (crash failures).

To make the proof easier to follow, the correctness proof of the algorithm is based on crash failures. Thus, if a crashed node "recovers", it must re-enter the system as a new node. Extending the algorithm to the crash-recovery model is described in Section 5.2 in detail.

It is also assumed that the propose module does not deliver "impossible" proposals where two different members have a common terminal node. If this cannot be guaranteed, simple checks must be introduced, which are omitted in this work.

3.3 Data Structures

3.3.1 Proposal and Group Structure

Every node has an integer id. Every group of nodes has an id consisting of a set of Δ node ids (the terminal nodes of the group, see Section 1.3.3 for details).

A proposal, as defined in Section 1.3.4, is a structure containing the following fields:

- $Members$: A set of Δ group or node ids.
- $Weight$: The weight of the group as defined in Definition 1.
- *Connections*: A set of $\frac{\Delta(\Delta-1)}{2}$ connection structures. A connection structure must contain a set of two endpoints and a positive weight of the connection. Other information possibly needed by *make connection*, *cancel connection* or the propose module can also be stored in that structure.
- $Terminals$: A set of Δ terminal node ids. This set is also the id of the group.

A group structure has the same fields as a proposal and the following additional fields:

- $ParentID$: The group id of the parent group, if applicable.
- *LockedBy*: $\perp$, if the group creation process has finished on all terminal nodes of all the group's members (i.e. on all nodes that know about the group, see below). Otherwise, it contains the unique id of the NBAC PROPOSE_GROUP instance in the process of creating the group.

All fields are initially $\perp$. While proposals only contain group proposals, a group structure can also be used to store information about a single node, in which case $Terminals$ contains the set of only the node's id, $Weight$ is 0, and $Members$ and *Connections* contain the empty set.[1] Thus, nodes are treated as a special kind of group with one terminal node and no members. During this paper, we use the term *group* for composite groups (groups with members and Δ terminal nodes) as well as for general groups (including composite groups and the special single-node groups). It should be clear from context what type of group is meant.

Example This is an example illustrating the use of the data structures and the weight calculation of Definition 1. Assume the following connection costs for the graph in Figure 1.4:

[1] Actually, the value of $Terminals$ does not matter for nodes stored in group entries. It is, however, useful to have *gid* = *Group*[*gid*].$Terminals$ to be consistent with Section 1.3.3.

Connections	Weight/Cost
$1-2, 2-3, 3-1,$ $4-5, 5-6, 6-4,$ $8-9, 9-10, 10-8$	1
$2-4, 3-7, 5-7$	2
$6-8, 7-12$	2.5
$12-9$	0.1
others	very high

This would result in the following example data structures:

	Group A	**Group E**
Members	$\{\{1\},\{2\},\{3\}\}$	$\{\{1,6,7\},\{12\},\{8,9,10\}\}$
Weight	$(3,\{\{1\},\{2\},\{3\}\},$ $\{1-2,2-3,3-1\})$	$(6+\epsilon,\{\{1,6,7\},\{12\},\{8,9,10\}\},$ $\{6-8,7-12,12-9\})$
Connections	$\{(\{1,2\},1),$ $(\{2,3\},1),$ $(\{3,1\},1)\}$	$\{(\{6,8\},2.5),$ $(\{7,12\},2.5),$ $(\{12,9\},0.1)\}$
Terminals	$\{1,2,3\}$	$\{1,10,12\}$
ParentID	$\{1,6,7\}$	$\{11,12,13\}$

Note that the (first component of the) weight of Group C is $\max(2+2+2, \max(3,3,0)+\epsilon) = 6$. Thus, the weight of Group E is $\max(2.5+2.5+0.1, \max(6,3,0)+\epsilon) = 6+\epsilon$.

Note that all but the first component of *Weight* is redundant. Thus, in the algorithm, for sake of simplicity, we will assume that *Weight* only stores the first component of the *real* weight and the member/connection information is implicitly used during comparisons. This suffices as the only purpose of the last components of the weight definition is to ensure unique weights of all groups.

3.3.2 Group Map

The main data structure used by the algorithm is the *Group* associative array. Its keys are general group ids (i.e. sets of one or Δ node ids); the values (group entries) are group structures. An example representing the overlay graph of Figure 1.4 can be found in Figure 3.1.

Note that, as this is a distributed algorithm, not every node needs every piece of group information, i.e. the contents of the *Group* associative array differ on different processors. The intuitive main motivation—and the mechanism used in previous versions of this algorithm—is that some *leader* of a group, e.g. the terminal node with the smallest id, must have all information about a group so that it can make decisions for this group. Note that, using this leader definition, a terminal node p of a group X, which is not leader of X, might become leader of a higher level group Y, with X being a member of Y (because some other node $q < p$, which was terminal node in X, builds an internal connection in

Y, thereby forfeiting its terminal node state within the higher level group Y). Thus, p must know about Y to (1) determine whether it is leader of Y and, if it is, to (2) make decisions on behalf of Y. To fulfill this requirement, the algorithm ensures that all terminal nodes of all members of a group *gid* have $Group[gid] \neq \perp$. However, as soon as the need arises to inform all terminal nodes of all members, we might as well drop the intuitive requirement for a leader and have all terminal nodes of all members of a group make decisions for that group. Thus, the algorithm does not use leader-based decision making any more. See Figure 3.2 for an example based on Figure 1.4.

	$gid = Group[gid].Terminals$	$Group[gid].Members$	$Group[gid].ParentID$
	$\{1\}$	$\{\}$	$(A)\{1,2,3\}$
	$\{2\}$	$\{\}$	$(A)\{1,2,3\}$
	$\{3\}$	$\{\}$	$(A)\{1,2,3\}$
	$\{4\}$	$\{\}$	$(B)\{4,5,6\}$
	$\{5\}$	$\{\}$	$(B)\{4,5,6\}$
	$\{6\}$	$\{\}$	$(B)\{4,5,6\}$
	$\{7\}$	$\{\}$	$(C)\{1,6,7\}$
	$\{8\}$	$\{\}$	$(D)\{8,9,10\}$
	$\{9\}$	$\{\}$	$(D)\{8,9,10\}$
	$\{10\}$	$\{\}$	$(D)\{8,9,10\}$
	$\{11\}$	$\{\}$	$(F)\{11,12,13\}$
	$\{12\}$	$\{\}$	$(E)\{1,10,12\}$
	$\{13\}$	$\{\}$	$(F)\{11,12,13\}$
A	$\{1,2,3\}$	$\{\{1\},\{2\},\{3\}\}$	$(C)\{1,6,7\}$
B	$\{4,5,6\}$	$\{\{4\},\{5\},\{6\}\}$	$(C)\{1,6,7\}$
C	$\{1,6,7\}$	$\{\{1,2,3\},\{4,5,6\},\{7\}\}$	$(E)\{1,10,12\}$
D	$\{8,9,10\}$	$\{\{8\},\{9\},\{10\}\}$	$(E)\{1,10,12\}$
E	$\{1,10,12\}$	$\{\{1,6,7\},\{12\},\{8,9,10\}\}$	$(F)\{11,12,13\}$
F	$\{11,12,13\}$	$\{\{11\},\{1,10,12\},\{13\}\}$	$\perp$

Figure 3.1: *Group map for Figure 1.4*

3.4 Primitives

- *make connection* and *cancel connection* are provided by the underlying infrastructure.

- *get connection weight* returns the actual weight of an existing connection. This value must only be known on one of the connection's endpoints[2]

[2] For reasons of simplicity we assume that this value is known by the endpoint with the lowest node id.

Node	*Group*[...] is set on this node
1	$\{1\}, (A)\{1,2,3\}, (C)\{1,6,7\}, (E)\{1,10,12\}, (F)\{11,12,13\}$
2	$\{2\}, (A)\{1,2,3\}, (C)\{1,6,7\}$
3	$\{3\}, (A)\{1,2,3\}, (C)\{1,6,7\}$
4	$\{4\}, (B)\{4,5,6\}, (C)\{1,6,7\}$
5	$\{5\}, (B)\{4,5,6\}, (C)\{1,6,7\}$
6	$\{6\}, (B)\{4,5,6\}, (C)\{1,6,7\}, (E)\{1,10,12\}$
7	$\{7\}, (C)\{1,6,7\}, (E)\{1,10,12\}$
8	$\{8\}, (D)\{8,9,10\}, (E)\{1,10,12\}$
9	$\{9\}, (D)\{8,9,10\}, (E)\{1,10,12\}$
10	$\{10\}, (D)\{8,9,10\}, (E)\{1,10,12\}, (F)\{11,12,13\}$
11	$\{11\}, (F)\{11,12,13\}$
12	$\{12\}, (E)\{1,10,12\}, (F)\{11,12,13\}$
13	$\{13\}, (F)\{11,12,13\}$

Figure 3.2: *Who knows what in Figure 1.4*

and is used to detect changing connection weights (e.g. because of moving nodes).

3.5 Description

The algorithm consists of single threaded, event/message-driven code. Note that all blocks of code up to the next **wait** statment are executed atomically and non-interruptable.

3.5.1 Main Loop

The main loop (Figure 3.3) is an enhanced version of the base algorithm of Figure 1.5.

Group is initialized only with the single-node group of the node itself. New proposals created by the propose module of the node are distributed to the terminal nodes of all members of the proposed group. An atomic commitment routine is used to ensure that, even in the case of node failure, either all members join the proposed group or none does.

Periodically, each node must verify that the locally stored groups are still valid. Due to the nature of the algorithm, it can happen that the information stored on two different nodes becomes inconsistent.

Example Groups x and y are members of group A. Group y decides to leave group A and join group B, which does not include x as a member. Let p be a node knowing about x but not about y. After y has changed groups, p still thinks that x is a member of group A, although group A is “broken”, because

```
1   var $Group[\,]$
2
3   $Group[\{ID\}].Members \leftarrow \{\}$
4   $Group[\{ID\}].Weight \leftarrow 0$
5   $Group[\{ID\}].Connections \leftarrow \{\}$
6   $Group[\{ID\}].Terminals \leftarrow \{ID\}$
7   $Group[\{ID\}].ParentID \leftarrow \perp$
8   $Group[\{ID\}].LockedBy \leftarrow \perp$
9
10  loop
11      wait for incoming message
12      if received proposal $P$ from local propose module
13          initiate atomic commit PROPOSE_GROUP:
14              $participants$ = all terminal nodes of all $P.Members$
15              $data = P$
16      if received signal to check for broken groups
17          var $gid \leftarrow Group[\{ID\}].ParentID$
18          while $gid \neq \perp \wedge Group[gid].LockedBy = \perp$
19              initiate atomic commit CHECK_GROUP:
20                  $participants$ = all terminal nodes of all $Group[gid].Members$
21                  $data = Group[gid]$
22              $gid \leftarrow Group[gid].ParentID$
23      if received atomic commitment message
24          execute atomic commitment phase
```

Figure 3.3: *Main loop*

y left. The periodic group consistency check on each node ensures that such changes are detected and updated in the local data structures, thereby avoiding group change notification messages and any inconsistency problems that could occur if such messages were delayed. As a positive side-effect, this check also detects node crashes and changed weights of groups (see below). In the group hierarchy, the check starts directly above the node itself (line 17) and then travels "upwards" the topology graph (line 22, see Figure 1.4(c)). Note that detecting node crashes and changed group weights is necessary anyway, which is another reason why we chose to also detect "left" groups using this polling method rather than using an additional (possibly more efficient) notification via explicit message passing.

As NBAC is usually implemented as a multi-phase protocol, we must listen for messages starting the next phase of an ongoing atomic commitment and execute the corresponding phase. Note that, on each node, many atomic commitment sequences can be active and waiting for messages initiating their next phase at any given time. Thus, it is mandatory to proceed with the correct phase of the correct atomic commitment sequence in line 24 (or initiate a new sequence, if an initial atomic commitment message arrives).

Note that both of the atomic commitment variants (PROPOSE_GROUP and CHECK_GROUP) use the set of all terminal nodes of all members of a group as participants. Thus, Δ (all members are single-node groups) to Δ^2 (all members are composite groups) nodes participate in each atomic commitment problem. This is used to distribute the information about the join or leave to all these nodes. See Section 3.3 for an explanation as to why this is necessary.

3.5.2 Atomic Commitment Functions

```
function vote_PROPOSE_GROUP($group$)              /* local decision of atomic commitment */
    var $gid \leftarrow$ element $gid$ of $group.Members$ with ID $\in gid$
    if $want\_to\_join(gid, group)$
        return VOTE_COMMIT        /* better group found */
    else
        return VOTE_ABORT        /* current group is better */

procedure decision_PROPOSE_GROUP($result, group, nbac\_id$)        /* locally execute NBAC decision */
    var $gid \leftarrow$ element $gid$ of $group.Members$ with ID $\in gid$
    if $result =$ COMMIT
        if $want\_to\_join(gid, group)$        /* check again, something might have changed */
            join_group($gid, group$)
            $Group[group.Terminals].LockedBy \leftarrow nbac\_id$  /* wait until everyone finished building */

/* decision has been executed everywhere */
procedure finalize_ PROPOSE_GROUP($result, finalize\_result, group, nbac\_id$)
    var $gid \leftarrow$ element $gid$ of $group.Members$ with ID $\in gid$
    if is_locally_consistent ($group$) $\wedge$ $Group[group.Terminals].LockedBy = nbac\_id \wedge$ \
            $result =$ COMMIT        /* see in check_group below */
        if $finalize\_result =$ ABORT        /* node crash after decision */
            leave_group($gid$)
        if $finalize\_result =$ COMMIT
            $Group[group.Terminals].LockedBy \leftarrow \perp$

function want_to_join($gid, group$)                /* true, if member gid should join group */
    return $Group[gid] \neq \perp \ \wedge$ \
            $(Group[gid].ParentID = \perp \vee group.Weight < Group[Group[gid].ParentID].Weight) \wedge$ \
            $(Group[group.Terminals] = \perp \vee Group[group.Terminals].Weight > Group[gid].Weight) \wedge$ \
            $group.Weight > Group[gid].Weight$
```

Figure 3.4: PROPOSE_GROUP

The atomic commitment algorithm (Figure 3.4 for PROPOSE_GROUP and Figure 3.5 for CHECK_GROUP) uses the procedure *vote*... () to decide on the local vote (VOTE_COMMIT or VOTE_ABORT). After a common decision has been reached, *decision*... () is called on each participating node to execute the common decision. After *decision*... () has been executed on all participating nodes, *finalize*... () (if used) is called on all participating nodes. In addition to the original *result*, a *finalize_result* is returned to indicate whether some node was suspected after *decision*... () has been executed. As a global COMMIT decision requires all participating nodes to be alive and to send some kind of VOTE_COMMIT message, we can piggyback some data ("commit data", returned with the **return** statement from *vote*... ()) with that message and safely assume that the set union of the commit data values from all the participants can be made available as parameter *commit_data* to *decision*... (). This parameter's value is undefined if the global decision was ABORT. See Section 2.3 for a detailed description of these atomic commitment extensions.

```
function vote_CHECK_GROUP($group$)                /* local decision of atomic commitment */
    var $gid \leftarrow$ element $gid$ of $group.Members$ with ID $\in gid$
    if  is_locally_consistent  ($group$)
        return (VOTE_COMMIT, \        /* group consistent */
                ($Group[gid].Weight, \sum_{c \in group.Connections : ID = min(c.Endpoints)}$ get connection weight($c$)))
    else
        return VOTE_ABORT        /* group inconsistent */

procedure decision_CHECK_GROUP($result, group, commit\_data$)        /* locally execute NBAC decision */
    var $gid \leftarrow$ element $gid$ of $group.Members$ with ID $\in gid$
    if  is_locally_consistent  ($group$)
        if $result =$ COMMIT        /* update weight */
            $Group[group.Terminals].Weight \leftarrow$ calculate_weight($commit\_data, group$)
            if $Group[gid].Weight \geq Group[group.Terminals].Weight$
                leave_group($gid$)             /* weight consistency  violated  */
            else  if $Group[group.Terminals].ParentID \neq \perp \wedge$ \
                    $Group[group.Terminals].Weight \geq Group[Group[group.Terminals].ParentID].Weight$
                leave_group($group.Terminals$)        /* weight consistency violated */
        if $result =$ ABORT        /* leave broken group */
            if $Group[gid] \neq \perp \wedge Group[gid].ParentID = group.Terminals$
                leave_group($gid$)

function   is_locally_consistent  ($group$)                /* $group \cong Group[group.Terminals]$?  */
    return $Group[group.Terminals] \neq \perp$  $\wedge$ \
            $Group[group.Terminals].Members = group.Members \wedge$ \
            $Group[group.Terminals].Connections = group.Connections$

function  calculate_weight ($data, group$)                /*  calculate  actual  weight  of  group */
    var $group\_weights \leftarrow$ create set ($\forall d \in data : d[0]$)
    var $connection\_weight\_sums \leftarrow$ create set ($\forall d \in data : d[1]$)
    return ($\max(\sum connection\_weight\_sums, (\max group\_weights) + \epsilon), group.members, group.Connections$)
```

Figure 3.5: CHECK_GROUP

PROPOSE_GROUP The nodes only vote VOTE_COMMIT if the received proposal is better than the current parent group. After a COMMIT group decision has been reached, all participants call *join_group* to update their internal data, if the member group has not been destroyed or joined a better proposal in the meantime (which can happen because atomic commitment group decision messages need not arrive in the correct order). We ensure that the weight of the new parent group is greater than the weight of the member (which is one of the requirements specified in Definition 1).

LockedBy is reset in *finalize* to ensure that the new group is only checked in the main loop after all participants have finished joining it. Otherwise, a CHECK_GROUP could destroy a perfectly fine group that has just not finished being built yet.

CHECK_GROUP VOTE_COMMIT is only returned if the group is consistent with the group entry of the atomic commitment initiator. If the node decides to ABORT, all members that still exist and have not left the group already leave it. If the participants decide to commit, the group weight is recalculated based on Definition 1 (implemented as function *calculate_weight*()).

3.5.3 Joining and Leaving Groups

```
procedure join_group($gid, group$)          /* member gid joins new group */
    if $Group[gid].ParentID \neq \perp$
        leave_group($gid$)
    $Group[group.Terminals] \leftarrow group$
    $Group[gid].ParentID \leftarrow group.Terminals$
    for all $c$ in $group.Connections$
        if ID is endpoint in $c$
            make connection $c$

procedure leave_group($gid$)          /* member gid leaves its current group */
    if $Group[gid].ParentID \neq \perp$
        for all $c$ in $Group[Group[gid].ParentID].Connections$
            if ID is endpoint in $c$
                cancel connection $c$
        leave_group($Group[gid].ParentID$)
        $Group[gid].ParentID \leftarrow \perp$
```

Both procedures assume that $Group[gid] \neq \perp$, which is checked in *decision*... ().

join_group() leaves the current group first, if necessary. Then, the group entry is modified, and connections in the underlying infrastructure are being created.

leave_group() removes the connections built by the group and clears unnecessary data structures. It requires that $Group[Group[gid].ParentID] \neq \perp$. The calling procedures (*join_group*(), *decision*... ()) will only guarantee that $Group[gid].ParentID \neq \perp$. However, Theorem 6 will prove that those two assertions are equal.

3.6 Improvements

This section presents extensions to the algorithm improving either complexity or implementability.

3.6.1 Leader-based Voting

Although the algorithm exhibits guaranteed convergence, one drawback is the extensive use of the potentially expensive atomic commitment protocol and, therefore, a potentially high message complexity (depending on the concrete system model). One idea, which has yet to be analyzed in detail regarding usefulness and implementability, is *leader-based voting*: A designated leader makes decisions on behalf of the group. This would reduce the maximum number of atomic commitment participants from Δ^2 to Δ. The following intuitive approaches to this problem can be identified:

1. The leader is a function of the set of terminal nodes, e.g. the terminal node with the smallest id. The set of terminal nodes, however, can change completely when moving up the hierarchy tree. For example, let Δ be 2 and let w.l.o.g. a_1 and b_1 be the leaders of the groups $\{a_1, a_2\}$ and $\{b_1, b_2\}$. If the best proposal happens to be one where a_1 and b_1 build an internal connection and a_2 and b_2 are the new terminal nodes, we have the situation that two nodes, that have not been leaders of a lower-level group, become leaders of a higher-level group. This introduces the problem that either: (a) non-leader terminal nodes must be informed about every decision leaders take or (b) the situation can arise that within a high-level group there is no terminal node knowing information about a complete path from this group down to one of the single-node groups. This would require some of the safety properties, e.g. the fact that there is no loop in the hierarchy, to be guaranteed in a distributed fashion rather than as a node-level invariant.

2. The leader is some inner node p of the group A, i.e. not necessarily a terminal node. This would allow to retain the failure-locality property, because if p crashes, all higher-level groups including A will be detroyed and, therefore, the loss of information does not do any additional damage. Additionally, using a suitable algorithm, it could be guaranteed that only a node that has already been leader in a member group of A can become leader in the higher-level group A, thus avoiding the problems mentioned above.

   The drawback, however, of this approach is the fact that the current composition of the *Group* map does not "know" about any nodes but the terminal nodes of the members and including a complete list of nodes into a group entry would exceed its bounded memory usage of $O(\Delta^2 \log n)$[3]. Thus, the need for an additional *Leader* field would arise.

3. Another option would be to use a designated node not necessarily within the group as data storage. This could even cause the atomic commitment requirement to be dropped, if all information is concentrated on a single node. However, this would completely eliminate fault-locality, thereby contradicting one of the main goals of this topology construction method.

3.6.2 Silent Atomic Commitment Participants

As Section 5 will show, ensuring a majority of correct participants for every atomic commitment instance can reduce the system model requirements. For example, the algorithm might still be implementable with fair lossy links or weaker failure detectors if this requirement is satisfied.

However, ensuring a majority of correct participants for every Δ-size subset of Π requires that the number of faulty nodes is at most $\lceil \frac{\Delta}{2} \rceil - 1$. As this is a very restrictive requirement, this extension works by including all nodes in Π in the NBAC rather than only the designated participants in each atomic commitment instance. These additional "silent" participants just vote VOTE_COMMIT. This change has the following effects:

- Message complexity increases.
- The probability of an outcome ABORT because of a suspected node increases.
- Requiring a majority of correct participants for each atomic commit instance allows $\lceil \frac{n}{2} \rceil - 1$ rather than $\lceil \frac{\Delta}{2} \rceil - 1$ nodes to fail.

[3] $O(\Delta^2)$ is the upper bound for the number of node ids in the individual fields of the group entry and $\log n$ is the maximum amount of memory consumed by one node id.

4 Correctness

A *stable period* is a period in time during which no node crashes or is suspected as faulty, no new nodes are added and no connection weights change.

For each stable period the underlying communication network can be seen as a fully-connected graph G with n' *regular nodes* Π' and n'' *gateway nodes* Π'' ($\Pi = \Pi' \cup \Pi''$ and $n = n' + n''$). See Section 1.3.2 for an extensive definition of the difference between regular and gateway nodes. According to Theorem 2, there is a unique overlay graph G' which our algorithm is supposed to create. As mentioned in the theorem, we require that $n' \geq 1$ and $n'' \geq 2\Delta - 2$. Note that if these requirements are not fulfilled, the algorithm does not lose liveness but simply does not produce the desired overlay graph. Thus, if the requirements are fulfilled during a later stable period which lasts long enough, the algorithm will produce the desired overlay graph then.

To show that the algorithm is correct, it will be proven that:

- At the beginning of each iteration of the main loop the local data structures are consistent.
- The algorithm does not deadlock.
- If the stable period lasts long enough, G' will eventually be constructed.

The following definitions are used (p being a node id):

- $Group_p$ is the local variable $Group$ at node p.
- $Parent_p^i$ is the "i-th parent of p". Formally, it is recursively defined as follows:
  - $Parent_p^0 := p$
  - $Parent_p^n := Group_p[Parent_p^{n-1}].ParentID$
- $Parentlist_p$ is defined as the list $(Parent_p^0, Parent_p^1, \dots, Parent_p^n)$, with $Parent_p^n \neq \perp$ and $Parent_p^{n+1} = \perp$.

4.1 Consistency

Lemma 1. *The assignment*
var gid $\leftarrow$ *element gid of group.Members with ID* $\in$ *gid*
always returns exactly one value for gid.

Proof. The assignment occurs at the beginning of atomic commitment functions. *group* is passed as a parameter of the atomic commitment. Thus, node ID participates in an atomic commitment with *group* being the data passed by the initiator. Assume by contradiction that no value for *gid* is returned, i.e. ID is not a terminal node of any of *group*'s members. However, lines 13 and 19 show that all atomic commitment participants are terminal nodes of at least one member of *group*. A contradiction.

We now assume that more than one candidate values for *gid* exist. This contradicts our assumption in Section 3.2 that no impossible proposals (i.e. proposals where two members share a common terminal node) are created. As the $Member$ field of group entries is never changed, this property also holds for the group entry passed to the CHECK_GROUP atomic commitment. □

Theorem 6. *For all nodes p: At the beginning of each iteration of the main loop (i.e. every time the execution reaches line 11), the following invariants hold* $(m, g \neq \perp)$*:*

1. *The node only knows about all groups in its parent list.*
   $\forall g : g \in Parentlist_p \Leftrightarrow Group_p[g] \neq \perp$

2. *p is a terminal node in all parent list entries except for the last one.*
   $\forall i, 0 \leq i \leq |Parentlist_p| - 2 : p \in Group_p[Parent_p^i].Terminals.$

3. *The weights of the parent list entries are strictly increasing.*
   $\forall m, g : Group_p[m].ParentID = g \Rightarrow Group_p[g].Weight > Group_p[m].Weight$

4. *The parent list is finite.*
   $\exists i : Parent_p^i = \perp$

5. *The parent-member relationship is consistent.*
   $\forall m, g : Group_p[m].ParentID = g \Leftrightarrow m \in Group_p[g].Members \wedge p \in m$

Note that, as *LockedBy*, *ParentID* and $Weight$ are the only fields that are changed in the algorithm, $Group_p[g] \neq \perp$ implies that all other fields of $Group_p[g]$ contain all the information initially provided by the proposal that was assigned to $Group_p[g]$ in *join_group*().

Proof. Exeution of the algorithm on a node can be seen as a series of computation events, each corresponding to an iteration of the main loop as one atomic step (until line 11 is reached again, which allows receive events to occur). Thus, we can prove this lemma by induction on the series of configurations of the node.

Section 3.5.1 shows that, initially,

- $Group_p[p] \neq \perp$
- $Group_p[p].ParentID = \perp$, and, thus, $Parentlist_p = (p)$

- $Group_p[p].Members = \{\}$
- $\forall q \neq p : Group_p[q] = \perp$

Thus, initially, the five invariants hold.

For the induction step, we analyze the following code blocks in which the *Group* array is changed and which could theoretically violate the invariants. Note that we can ignore changes to *LockedBy* as they do not affect the invariants. The only code blocks to be considered are the *decision...*() and *finalize...*() functions (and the subroutines they call). By showing that these functions do not violate the invariants, we prove the correctness of the theorem.

Note that *leave_group*(*gid*) requires $Group[gid] \neq \perp$. To do something useful, $Group[Group[gid].ParentID] \neq \perp$ must hold as well. This is, however, satisfied at all code lines where *leave_group*() is called directly. (The case where *leave_group*() is called from *join_group*() is handled further below.) Note that, in the code lines mentioned below, Lemma 1 guarantees that *gid* contains a valid value which, by definition, is in $group.Members$.

- Line 45: *is_locally_consistent*() guarantees that $Group[group.Terminals]$ exists and its $Members$ field is consistent with $group.Members$. Thus, *gid*, a member of $group.Members$, is also a member of $Group[group.Terminals].Members$. The fifth invariant thus implies that $Group[gid]$ exists as well and $Group[gid].ParentID = group.Terminals$.
- Line 68: Analogous to line 45.
- Line 71: *is_locally_consistent*() guarantees that $Group[group.Terminals]$ exists. The **if** statement right before line 71 assures that $Group[group.Terminals].ParentID \neq \perp$ and, using the fifth invariant, thus $Group[Group[group.Terminals].ParentID] \neq \perp$ as well.
- Line 74: Analogous to line 45.

From this we can conclude that, if the first invariant was satisfied before calling *leave_group*, the parameter passed to *leave_group* is in $Parentlist_p$.

check_group The recursion in *leave_group* sets the group entries of all entries in $Parentlist_p$ following (but not including) *gid* to $\perp$. $Group[gid].ParentID$ is also set to $\perp$, making *gid* the last entry in $Parentlist_p$. Thus, the first invariant still holds after execution of *leave_group*().

As no new entries are added to $Parentlist_p$, the second and forth invariants are still satisfied. If no group weights are changed (line 66), the same holds for the third invariant. Otherwise, assume that the third invariant is violated in line 66. There are two possibilities: $Group[group.Terminals].Weight$ can become smaller or equal to weight of the previous parent list entry (*gid*). This is checked in line 67 and corrected in the following line by removing all entries following *gid* from the parent list. The other case, $Group[group.Terminals].Weight$ becoming greater or equal to its parent, is checked in line 69 and corrected in the following line by removing all entries following *group.Terminals* from the parent list.

To prove the fifth invariant it suffices to look at the group entries contained in $Parentlist_p$, as the first invariant shows that these entries are the only ones having $ParentID$ or $Members$ set. Let i be the index of *leave_group*'s parameter *gid* in the parent list. The group entries from $Parent_p^0$ to $Parent_p^{i-1}$ did not change at all (as the third invariant proves that there are no duplicate entries in the parent list) and therefore still satisfy the fifth invariant. $Group[Parent_p^{i-1}].ParentID = Parent_p^i$, $Parent_p^{i-1} = Group[Parent_p^i].Members$ and $ID \in Parent_p^{i-1}$ also hold because those values have not been changed. $Group[Parent_p^i].ParentID$, the $ParentID$ of the last element in the parent list, equals $\perp$, thereby also satisfying the fifth invariant.

propose_group We have just shown that *leave_group*() does not violate the invariants if it is guaranteed that its parameter *gid* satisfies $Group[gid] \neq \perp$ and $Group[Group[gid].ParentID] \neq \perp$. *join_group*'s parameter *gid* fulfills the first requirement, which is guaranteed in function *want_to_join*(). The second requirement is checked in line 86 (using the first invariant). Thus, the call to *leave_group*() in line 87 returns with a consistent group structure. Note that after line 87, *gid* is the last entry in the parent list, because either line 86 evaluated to false or *leave_group*(*gid*) has been called.

Afterwards, $Group[group.Terminals]$ is set and $Group[gid].ParentID$ is linked to this new group, making it the new last entry in the parent list.

$Group[group.Terminals].ParentID$ must be $\perp$ (thus ending the parent list), because proposals always have $ParentID = \perp$. This is consistent with the first invariant. The second invariant holds because of the definition of *gid* in line 33 by ensuring that the node id ID $(= p)$ is in *gid*, the new second-to-last entry in the parent list.

There are two ways to violate the third invariant. The first one occurs if the newly joined group has a lower (or equal) weight compared to its member *gid*. However, function *want_to_join*(), which is called right before *join_group*(), ensures that the new group's weight is strictly greater than the weight of *gid*. The second possibility to violate the third invariant is by changing the weight of an existing group. This can happen if the new group's id is already present in the parent list (i.e. $Group[group.Terminals] \neq \perp$) and, thus, the group entry gets overwritten in line 88. Let us look at the state of the node right before calling *join_group*(). Let i be the index of the member that wants to join the new group. We have two cases:

1. $group.Terminals \in (Parent_p^0, \ldots, Parent_p^i = gid)$. Then we know that *want_to_join*() evaluated to true, that $Group[group.Terminals] \neq \perp$ and, therefore, $Group[group.Terminals].Weight > Group[gid].Weight$ (see

line 52). This, however, contradicts our assumption that the third invariant was satisfied before calling *decision*_PROPOSE_GROUP() (i.e. at the beginning of the main loop iteration).

2. $group.Terminals \in (Parent_p^{i+1}, \ldots, Parent_p^{|Parentlist_p|-1})$. This does not violate the invariant either because in the new parent list after executing *join_group*() *group.Terminals* is the $i+1$-th entry, ending the list.

The third invariant thus guarantees that no loop exists in the parent list. As every statement can add at most one entry to the parent list, the forth invariant is satisfied as well.

We have already shown that after the call to *leave_group*() the invariants are still fulfilled. Note that after line 88 $Group[group.Terminals].ParentID = \perp$ and, therefore, does not need to be considered. Thus, we only have to show that after *join_group*() $gid \in Group[group.Terminals].Members$ and $p \in gid$. Note that $Group[group.Terminals] = group$. The definition of *gid* in line 33 thus gurantees that the fifth invariant is not violated. □

4.2 Liveness

Lemma 2. *All incoming messages will eventually be processed.*

Proof. To prove this lemma we must show that no operation executed in the main thread blocks the thread forever. There is one recursion and one loop that need to be analyzed:

First, there is the recursion in *leave_group*(). We have shown in the proof of Theorem 6 that, when *leave_group*() is called, $Group[gid] \neq \perp$. Thus, according to Theorem 6, $gid \in Parentlist_p$. The recursion traverses the parent list until it reaches the end ($Group[gid].ParentID = \perp$). As the parent list is finite, the end is eventually reached and *leave_group*() terminates. As there are no changes to the parent list from the start of the main loop iteration to the point right before *leave_group* is called, we can use the above invariants.

Then, there is the **while** loop in line 18. The loop traverses the parent list from the second to the last entry, unless it is terminated earlier by an unfinished group. As above, we can use the fact that the parent list is finite to prove the termination of this loop.

All other loops are **for** loops on finite sets and, therefore, do not pose a problem, as the sets are not modified within the loop body. Thus, all iterations of the main loop eventually terminate, and all messages in the message queue will eventually be received in line 11. □

Lemma 3. *All atomic commitment sequences eventually terminate.*

Proof. As the atomic commitment protocol itself is assumed to be non-blocking this follows directly from the the previous proof that the execution cannot get stuck in one of the subroutines. □

4.3 Convergence

Even with the safety and liveness properties from the previous sections, it is not at all obvious that the algorithm will converge. This section proves formally that the unique overlay graph will eventually be constructed, provided that (a) a perfect propose module is used and (b) at every node, the check group signal occurs infinitely often in infinite runs.

4.3.1 Proof Idea

The reader familiar with [TS04] might argue that a convergence proof already exists in that paper. However, that proof, which is repeated here for convenience, is not enough to prove convergence of the algorithm presented in this work.

Theorem 7. *The presented algorithm with a perfect propose module converges and constructs the unique minimum admissible overlay graph G' for every communication graph G with $n'' \geq 2\Delta - 2$, provided that G remains stable.*

Proof. We use induction on the number $i \geq 0$ of groups in the set G_i of stable groups constructed so far. For $i = 0$, $G_0 = \Pi$ contains all single-node "groups" only. Eventually, the perfect propose module generates the minimal proposal with $P.Members \subseteq G_0$ and $g_0 = P.Terminals$[1]. From the algorithm, it follows that every node in $P.Members$ accepts the proposal and joins the group. After all, there is no alternative group containing a node from $P.Members$ with less weight, since (1) g_0 is minimal and each higher-level group that contains g_0 has higher weight than g_0 according to Definition 1. Therefore, the group g_0 is stable in that it will not be destroyed again later on.

For $G_i = G_{i-1} \cup \{g_{i-1}\}$, eventually some propose module generates the minimal proposal with $P.Members \subseteq G_i$ and $g_i = P.Terminals$. For the same reasoning as for $i = 0$, the group g_i is eventually joined by every member in $P.Members$ and remains stable. The algorithm hence converges from the bottom to the root of the topology tree. Since Theorem 1 holds also for the minimum admissible overlay graph, it is ensured that all regular nodes are used up before gateway nodes are considered. Hence, the root group can be constructed since at least $2\Delta - 2$ gateway nodes are available. □

(Note that this proof uses g_i in a different way than the proof presented in the next section.)

[1] Recall that the set of terminal node id's $P.Terminals$ is used as the group id.

Although this proof gives some important insights into the general behavior of the construction method, it cannot be applied directly to the presented algorithm. Consider the topology from Figure 4.1 as a counterexample for $\Delta = 3$:

Figure 4.1(a) contains the underlying communication graph. We are assuming a fully-connected graph; thus, all edges not shown in this figure are assumed to have very high cost. Note that $\{1, 2', 3'\}$ and $\{a, b, c\}$ are the two smallest groups in G' (see Figure 4.1(b)):

- $\{1, 2', 3'\}$, because it is the overall group with smallest weight and, therefore, if a proposal for $\{1, 2', 3'\}$ arrives, it will be accepted;

- $\{a, b, c\}$, because after $\{1, 2', 3'\}$ has been created (which will eventually happen, given a perfect propose module), the low-weight group $\{1, 2, 3\}$ cannot be built any more and, therefore, $\{a, b, c\}$ is the next smallest group.

Thus, $\{1, 2', 3'\}$ corresponds to g_0 in the above theorem and $\{a, b, c\}$ corresponds to g_1.

Consider the situation depicted in Figure 4.1(c): There have been proposals for the groups $\{1, 2, 3\}$, $\{1, 4, 5\}$ (members $\{1, 2, 3\}, 4, 5$) and $\{5, a, b\}$ (members $\{1, 4, 5\}, a, b$), which have all been accepted. Any proposal for g_1 would be rejected by a and b because they would not want to leave a group with weight $41 + \epsilon$ to join a group with weight 45.

Now a proposal for g_0 arrives, which is accepted. The theorem claims that if a proposal for g_1 arrived now, it will be accepted eventually. Note, however, that this is not at all obvious, as a and b still believe that they are part of the "old" group $\{5, a, b\}$ with weight $41 + \epsilon$ and, thus, reject proposals for g_1. Fortunately, node 1 knows that $\{5, a, b\}$ no longer exists, and it is one of the nodes included in the next CHECK_GROUP for group $\{5, a, b\}$, causing the information to propagate upwards quite soon. A more complex example, however, could include additional layers of groups between $\{1, 2, 3\}/\{1, 2', 3'\}$ and $\{5, a, b\}$, making it necessary for the information to travel upwards the topology tree though well-placed CHECK_GROUPs, until g_1 can finally be accepted.

Consider another example: Starting with the situation in Figure 4.1(c), proposals for $\{1, 2', 3'\}$, $\{a, b, c\}$ and $\{1, 4, 5\}$ cause the overlay graph to exhibit the structure of Figure 4.1(d). A delayed proposal for group $\{5, a, b\}$, which has been created when the graph looked like Figure 4.1(c) and thus incorrectly states the weight of the group as $41 + \epsilon$ rather than $51 + \epsilon$, could cause the existing group g_1 with weight 45 to be destroyed, until, eventually, the weight information is updated through CHECK_GROUPs. Again, by introducing additional layers into the example it can be shown that more than one CHECK_GROUP might be necessary before convergence of the algorithm can resume.

Thus, we can conclude that Theorem 7 does not suffice to prove convergence of the topology construction algorithm, since the fact that g_i will eventually

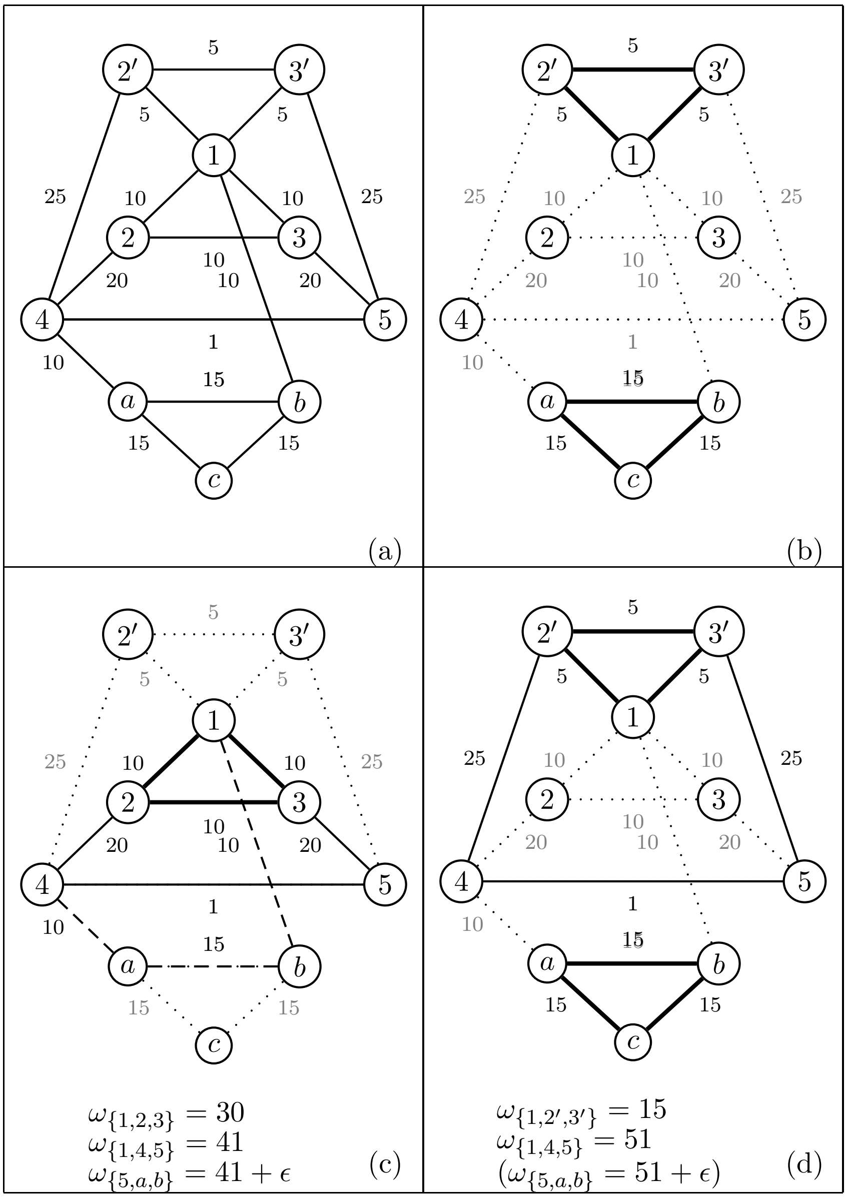
$2'$
$3'$
1
2
3
4
5
a
b
c
5
25
10
20
1
15
(a)
(b)
(c)
(d)
$\omega_{\{1,2,3\}} = 30$
$\omega_{\{1,4,5\}} = 41$
$\omega_{\{5,a,b\}} = 41 + \epsilon$
$\omega_{\{1,2',3'\}} = 15$
$\omega_{\{1,4,5\}} = 51$
$(\omega_{\{5,a,b\}} = 51 + \epsilon)$

Figure 4.1: *Example topology*

be joined is not proven (although one might intuitively expect—and we will show—that this must eventually happen).

4.3.2 Full Proof

In what follows, we will assume that, in addition to a perfect propose module, it will be the case that infinitely many check group signals will occur at every node in an infinite run.

We use the following definitions:

- An atomic commitment sequence is *finished* iff *finalize_*...() (or *decision_*...(), if *finalize_*...() is not employed), has been executed on all participating nodes.
- An atomic commitment sequence is *active*, iff it has been initiated, but it is not *finished* yet.

Note that neither a group id nor a group entry can uniquely identify a "group" in its topological structure. A group entry (i.e. a local entry in the *Group* map) only stores information about the member ids of the group, not about the composition of these members. Thus, it is possible that two group entries are equal but the underlying trees below the group's members have different structure.

Fix any point in time as the beginning of the stable period. This stable period ends if any node crashes, any node is added to the system, any connection weight changes or the underlying failure detector of the atomic commitment protocol suspects a node to have crashed.

We can now define the convergence of the algorithm as follows: If the stable period lasts long enough, eventually the unique overlay graph will be constructed and all groups will be stable, in the sense that they will not be destroyed during the stable period.

Let $C = (g_1, \dots, g_m)$ be the list of all general (i.e. single node and composite) groups in G', the unique overlay graph as defined in Theorem 2, in the order of increasing weights. Let C_i be the i-element prefix set of C. Note that C_n is the list of all 0-weight single-node groups (in arbitrary order). We define:

Definition 5. *A group entry is i-*weighted*, if*

- *it has a corresponding group g in C_i with the same group id and the same weight or*
- *it has a weight greater than the weight of g_i.*

*A group id gid is i-*stable *at a given time, if, in the current configuration and every following configuration until the end of the stable period, all group entries on all nodes with this id are i-weighted.*

A group id exists permanently *if it will not be destroyed until the end of the stable period.*

Informally, an i-weighted group entry is a group entry which will not prevent the creation of group g_i. See Figure 4.2 for an overview of the convergence proof. We show that if all groups are $i-1$ stable and the groups $g_0, \ldots, g_{i-1}$ exist permanently, there exists a time t_1 after which the information in the propose modules is sufficiently accurate. (Definition 6 explains the exact meaning of *sufficiently accurate* in this case.) Then, an inner induction shows that there exists a time t_2 after which all group ids are i-stable, i.e. all group entries on all nodes are i-weighted. Afterwards, we can show that the claim from Theorem 7 holds and g_i will be constructed and cannot be destroyed until the end of the stable period.

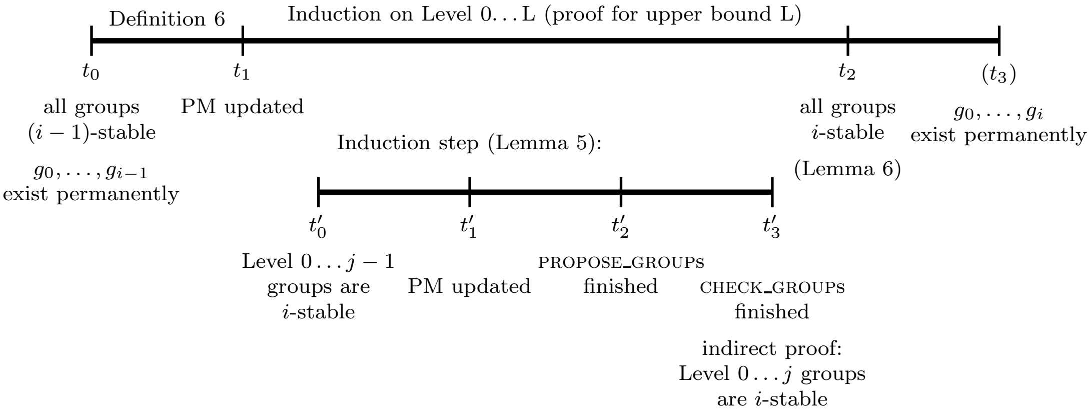

Figure 4.2: *Overview of the convergence proof*

Lemma 4. *For all i, $n \leq i \leq |C|$ it holds: If the stable period lasts long enough, there is a time after which all groups in C_i exist permanently and all group ids are i-stable.*

Proof. Initially, for C_n the lemma holds trivially: All single-node groups $g_1, \ldots, g_n$ always exist and all composite groups have (and will always have) a weight greater than 0, the weight of g_n, the last single-node group.

It can be shown that, if the condition is satisfied for $i-1$ at time t_0, eventually, it is satisfied for i, too. As, by assumption, all groups in $(g_1, \ldots, g_{i-1}) = C_{i-1}$ exist permanently and their group ids are $i-1$-stable, these ids are by definition also i-stable. Thus, we just have to show that eventually g_i is constructed and becomes i-stable, and that all other group ids become i-stable as well.

Definition 6. *Let t_1 be the time after t_0 when all propose modules have adapted to the change (i.e. they know the correct connection weights and if they propose a group where all member group ids are in C_{i-1}, the proposal has correct weight).*

We will now show that there is a time $t_2 > t_1$ after which all group entries on all processors have either correct weight, if their group id is in C_i, or have a weight greater than the weight of g_i (which might not have been built yet). To do this, we introduce the concept of the (time-variant) *level* of a group entry. A single-node group always has level 0. For composite groups, the definition is a bit more complicated. The intuitive notion would be to define the level as follows:

Informally, the level corresponds to the height of the group in the topology tree. As we cannot guarantee global consistency of the group entries, the level is "attached" to the group entry during atomic commitment, where there is agreement between the different members of a group. We use $level'$ rather than $level$ in the formal definition below because this intuitive definition is not sufficient, as we will show.

Definition 7. *The* $level'^t_p[gid]$ *of a group id gid on a node p at time t is defined as*

- 0 *for single-node group ids and*
- *at each successful* CHECK_GROUP *or* PROPOSE_GROUP *with participant set* P*, the level is updated to* $1 + \max_{p' \in P} level'^{t_{p'}}_{p'}[m_{p'}]$*, with* $m_{p'}$ *being the member of gid on whose behalf* p' *participates in the* CHECK_GROUP *or* PROPOSE_GROUP *and* $t_{p'}$ *being the time at which* p' *voted* VOTE_COMMIT.

Now we would use an induction proof on the level of the group ids to show that all group ids are i-stable. Note, however, that the induction step $(j-1 \rightarrow j)$ would fail in the following case: The group entries for group A have level′ $k > j$. Propose module X knows about group A and proposes a new group B with A as one of its members. In the meantime, however, the graph reconstructs itself, group A is destroyed and, later, a group A' with the same terminal nodes (and, thus, the same group id) as A is built. Assume that a group entry of A' has level′ $j-1$, all other members of B have level′ less than or equal to $j-1$ and group B is built when the proposal arrives at the nodes. (Note that the nodes do not see a difference between the proposed member A and A', as the $Member$ field of a proposal only contains group ids.) Until the next CHECK_GROUP for B is performed, we cannot use the induction hypothesis to prove that B's weight satisfies the i-stable requirement, because the propose module calculated its initial weight using the weight of A whose group entries level′'s were $k > j$.

There are two ways to solve this problem: One is to prove that there is an upper bound b for how often such a situation can happen and define level j group ids to be i-stable after b cycles of CHECK_GROUPs and propose module information updates. The other, which is used below, is to redefine the level as follows:

Definition 8. *The* $level^t_p[gid]$ *of a group id gid on a node p at time t is defined as* 0 *for single-node group entries. For a composite group entry gid,*

- *the level is set initially during the decision phase of* PROPOSE_GROUP *for a proposal from some propose module* X *to*

$1 + \max_{m \in Group_p[gid.Members]} level_{p_m}^{t'_m}[m]$ *with* t'_m *being the time at which the last update regarding* m*'s weight has been sent to* X *that has been received before* X *sent out the proposal.* p_m *is the node that sent this update to* X*.*

- *During the decision phase of each successful* CHECK_GROUP *with participant set* P*, the level is updated to* $1 + \max_{p' \in P} level_{p'}^{t_{p'}}[m_{p'}]$*, with* $m_{p'}$ *being the member of gid on whose behalf* p' *participates in the* CHECK_GROUP *and* $t_{p'}$ *being the time at which* p' *voted* VOTE_COMMIT*.*

Intuitively, this means that the level of a newly created group entry is the level the propose module "thought" that the new group entry would have. The process of updating the level is similar to updating the group weight during a successful CHECK_GROUP. Now we can use a straight-forward induction proof on the following lemma.

Lemma 5. *For all* $j \geq 0$ *holds: There is a time after which in all configurations until the end of the stable period all group entries whose level is less than or equal to level* j *are* i*-weighted.*

Proof. Proof by induction. Level 0 group entries (single-node groups) are always i-stable. Assume that after time t'_0 the lemma holds for $j-1$. Let t'_1 be the time after which the information in the propose modules has been updated for all group entries that existed at time t'_0. Let t'_2 be the time after all PROPOSE_GROUPs whose proposals were created before t'_1 have been finished. Let t'_3 be the time by which for every group entry that existied at t'_2 a CHECK_GROUP has been started after t'_2 and finished.

Finally assume by contradiction that there is a time t after t'_3 (but within the stable period) in which a group entry g with group id *gid* on node p has a level less than or equal to j but is not i-weighted. If the group entry has a level less than j, this contradicts the assumption that the lemma holds for $j-1$. If the group entry has level j, we have the following cases:

- The last update to the level of the group entry g has been a PROPOSE_GROUP. This PROPOSE_GROUP must have been created after t'_1 (otherwise, there would have been a CHECK_GROUP for this group entry between t'_2 and t'_3). Thus, by definition of t'_1, the proposal was based on group information sent from some nodes after t'_0. As the group entry has level j, its members had level $j-1$ or less when the propose module was updated (according to the definition of *level*). As the propose module was updated after t'_0, we can assume that all groups with level $j-1$ or less were i-weighted. Here we have two cases:
  - At least one member had a weight greater than the weight of g_i. Then, the weight of the proposed new group was also greater than g_i. As there has been no CHECK_GROUP in the mean-time, the weight

of g is still greater than g_i. Thus, g is i-weighted and we have the required contradiction.

  - All members were in C_i. As they were i-weighted, their weights were correct. Thus, we have the following cases: (1) g's id is in C_{i-1}. This contradicts the assumption of the outer induction that the groups in C_{i-1} already existed permanently before t'_1. (2) g's id is not in C_{i-1} and its weight is less than g_i's weight. This contradicts the minimality of g_i (as all members are in C_i). (3) g's id is not in C_{i-1} and its weight is equal to or greater than the weight of g_i. This contradicts the assumption that g is not i-stable. (Note that two groups with different group ids cannot have the same weight according to Definition 1.)

- The last update to the level of the group entry g has been a CHECK_GROUP. This CHECK_GROUP must have been started after t'_2. As the group entry has level j, the group entries of the members must have had level $j-1$ or less during the vote phases of their terminal nodes. By assumption we know that these entries were i-weighted and, thus, the same reasoning as in the previous point applies (except for the case where $g \in C_{i-1}$, this now leads to a contradiction regarding the assumption that g is not i-stable, as all groups in C_{i-1} permanently exist and the weight of the members must be correct).

□

However, Lemma 5 does not yet prove that i-stability for all nodes is reached in finite time as level numbers can rise without bound. We can, however, show that after a certain level a group entry must have weight greater than the weight of g_i, thereby becoming i-weighted. Note that for every given tuple (t, p, gid) which has a level $level^t_p[gid] > 0$ and a weight $Group_p[gid].Weight$ (at time t) associated to it, we can use the definition of $level$ to find a tuple (t', p', gid') with $level^{t'}_{p'}[gid'] = level^t_p[gid] - 1$ and a weight $Group_{p'}[gid'].Weight$ (at time t') $< Group_p[gid].Weight$ (at time t). Note that the difference between those weights is at least the fixed constant ϵ. Thus, we know that there must be an (unknown) level L, where all group entries with level L or greater must always have a weight greater than the weight of g_i. Using this fact and Lemma 5 we can conclude the following lemma:

Lemma 6. *There is a time t_2 after which all group ids are i-stable.*

It is easy to see that, if g_i does not exist yet and a proposal for g_i arrives, it will be accepted because all conditions in *want_to_join*() are satisfied:

$Group_p[gid] \neq \perp$: $gid \in C_{i-1}$ and, by assumption, all groups in C_{i-1} exist permanently.

($Group_p[gid].ParentID =\perp \lor group.Weight < Group_p[Group_p[gid].ParentID].Weight$): Assume by contradiction that there is some better parent group g' than the proposed g_i.

If $g' \in C_i$, we have a contradiction because two groups g' and g_i with the same member cannot coexist in G'. Thus, $g' \notin C_i$. Lemma 6 shows that in this case the weight of g' is greater than the weight of g_i, which contradicts the assumption that g' is better.

($Group[group.Terminals] =\perp \lor Group[group.Terminals].Weight > Group[gid].Weight$): Assume that this condition is violated. This would mean that g_i's terminals are already in the parent list below *gid.* As $gid \in C_{i-1}$, this would mean that g_i is also in C_{i-1}. This would introduce a duplicate entry in C, which is prohibited by construction.

$group.Weight > Group[gid].Weight$: *gid* is in C_{i-1} and, therefore, must have correct weight. After time t_2, all proposals where all members are in C_{i-1} have correct weight. Thus, this condition is satisfied.

We have shown that g_i is eventually being constructed after t_2. To show that g_i exists permanently, we assume by contradiction that g_i will be destroyed on some node p any time in the future (during the stable period). This could happen in the following cases:

- Some member m of g_i joins a better proposal P. As Lemma 6 guarantees that the group id of P is i-stable, P has either greater weight than g_i, which contradicts the assumption that P is better than g_i, or P is in C_i. However, g_i is also in C_i and there cannot be two groups with the same member in C.

- CHECK_GROUP returns ABORT for g_i. As we have shown that no member will join a better proposal, this can only happen if a member $m \in C_{i-1}$ is destroyed, which violates the induction assumption, or a processor is suspected to have crashed, which violates the condition that we are within the stable period, where no crashes or false suspicions are allowed.

- CHECK_GROUP returns COMMIT for a lower-level group but causes g_i to be destroyed because of a weight inconsistency. As the weights of the lower-level groups (which are in C_{i-1}) do not change and Lemma 6 assures that they are correct, this cannot happen.

□

Lemma 4 shows that the groups in G' will eventually be constructed. As G' is complete, i.e., builds a hierarchy from the single-node groups to high-level groups including the gateway nodes, there is no room for other groups without breaking the existing structure. As the groups exist permanently and thus cannot be destroyed as long as the stable period lasts, we can conclude the following theorem:

Theorem 8. *If the stable period lasts long enough, the constructed topology will be the unique overlay graph.*

5 Implementability

Section 3.2 stated the following system-level requirements for the topology construction algorithm: (1) a non-blocking weak atomic commitment service (see Section 2.1) and (2) a fully-connected asynchronous system (see Sections 1.2.2 and 1.2.3).

Agreement problems and their solvability in asynchronous systems has been studied extensively in literature. This chapter gives an overview over the current state of research and applies this information to the presented algorithm. Note that in contrast to previous parts of this work, where *atomic commitment* and *NBAC* were used as synonyms for the *non-blocking weak atomic commitment* problem, this section uses a more differentiated terminology: *atomic commitment (AC)*, *non-blocking atomic commitment (NB-AC)* and *non-blocking weak atomic commitment (NB-WAC)* and their respective unique abbreviations refer to three different problems as specified in Section 2.1. When following the references in this work, note that the terminology used in the papers is not consistent.

Intuitively the following relations hold between these problems: AC $\leq$ NB-AC and NB-WAC $\leq$ NB-AC.

[TS92] gives an introduction on agreement problems. They also claim that

> The commit problem is strictly harder to solve than the consensus problem because of this priority in favor of aborts. Therefore, any result indicating the impossibility of consensus translates to an impossibility result for the commit problem.[1]

Such an impossibility proof for a deterministic solution of the consensus problem in the presence of failures in a purely asynchronous system has been given by Fischer, Lynch and Patterson [FLP85]. Thus, AC is not solvable in the presence of failures. As any solution to the NB-AC problem would also solve AC, this also proves the impossibility of NB-AC.

5.1 Crash Failure Detectors

To circumvent this problem without restricting the asynchronous model more than necessary, Chandra and Toueg [CT96] introduced the concept of *unreliable failure detectors* (FDs). Local failure detector modules monitor the system

[1] Note that this claim is not entirely correct: Guerraoui showed in [Gue02] that NB-AC is actually *not* strictly harder than consensus. Nevertheless, the impossibility result holds.

and inform the algorithm about nodes they suspect to have failed. Chandra and Toueg identified multiple classes of failure detectors, distinguished by their accuracy properties:

- *Perfect* ($\mathcal{P}$): No process is suspected before it crashes.
- *Strong* ($\mathcal{S}$): Some correct process is never suspected.
- *Eventually Perfect* ($\diamond\mathcal{P}$): There is a time after which correct processes are not suspected by any correct process.
- *Eventually Strong* ($\diamond\mathcal{S}$): There is a time after which some correct process is never suspected by any correct process.

All of these have in common that eventually every process that crashes is permanently suspected by every correct process (completeness). [CT96] also presents a class of FDs satisfying only a weaker variant of the completeness property ("Eventually every process that crashes is permanently suspected by some correct process."), but as those FDs can be transformed into failure detectors satisfying the stronger property, we can ignore this variant for our analysis of solvability.

[CT96] also shows that consensus is solvable using $\mathcal{S}$ in any asynchronous systems; $\diamond\mathcal{S}$ suffices, if there is a majority of correct processes. In [Gue95], Guerraoui presents a reduction from NB-WAC to consensus; thus, the same results hold for NB-WAC. The question, whether a weaker failure detector would suffice for solving NB-WAC is raised in [Gue95] but left open for further research. While there are some papers on minimal FDs for NB-AC, for example [DGFG$^+$04] by Delporte-Gallet and Fauconnier, NB-WAC still lacks such results.

Recall that the *stable period* defined in Section 4 requires that no node is falsely suspected by any failure detector during the time necessary to construct the overlay graph. We can only *guarantee* this requirement using the FDs specified above by choosing an eventually perfect or perfect failure detector. Otherwise, with $\diamond\mathcal{S}$ or $\mathcal{S}$, it is possible that some group is never built because one of the nodes is permanently suspected by some failure detector. In practice, a weaker FD might suffice if false suspicions are seldom enough: If the frequency is low enough, convergence (i.e. construction of the overlay graph) will be delayed but not prevented. After the overlay graph has been constructed, a false suspicion during a CHECK_GROUP for some group *gid* will cause all groups in the group hierarchy on the path from *gid* to the root group to be destroyed. Due to the failure locality properties of the algorithm, other subtrees are not affected and the overlay graph will be rebuilt as soon as the corresponding proposals arrive. For example, a falsely suspected node 8 in Figure 1.4 could only cause groups D, E and F to be destroyed but would not affect groups A, B or C.

Thus, if we combine the requirements imposed by the need to implement NB-WAC with arbitrary many failures ($\mathcal{S}$) and the absence of false suspicions during the stable period ($\diamond\mathcal{P}$), the minimal failure detector that guarantees convergence in every case is $\mathcal{P}$.

Another possibility would be to implement the silent participant extension presented in Section 3.6.2: With this modified algorithm and the additional constraint that a majority of nodes is correct, $\diamond\mathcal{S}$ would suffice to implement NB-WAC. Thus, an eventually perfect FD ($\diamond\mathcal{P}$) could be used to satisfy all requirements and guarantee convergence.

Note, however, that $\mathcal{S}$ is not the proven weakest failure detector for implementing NB-WAC with an arbitrary number of failures and, therefore, it might well be possible that $\diamond\mathcal{P}$ suffices even without the need to modify the algorithm and the requirement for a majority of correct processes. Actually, we assume that it might be possible to devise an extension of the algorithm which tolerates non-terminating NB-WAC instances and still guarantees convergence. In that case, $\diamond\mathcal{P}$ would suffice without the silent participant extension even with a majority of faulty processes. See Section 6.1 for details on this approach.

5.2 Crash & Recovery

So far, the only failure possibility which has been considered for nodes in this work is the crash failure model: A node works correctly until some time t, after which no more messages are sent by this node. If a node "recovers", it enters the system as a new node with a new unique id and an empty state.

The *crash-recovery* model as specified in [ACT98] differentiates between

- two types of *good* processors: *always up* (never crashes), *eventually up* (will crash at least once but will eventually remain up) and
- two types of *bad* processors: *eventually down* (will eventually crash permanently), *unstable* (keeps on crashing and recovering infinitely many times).

Before presenting a solution for the crash-recovery model, we must define the behavior the algorithm should have in this model. Clearly, unstable nodes can prevent the creation of the overlay graph by participating when they are up and causing their groups to be destroyed again when they are down.

As the stable period already requires that no nodes crash and no new nodes are added (i.e. no nodes recover), it suffices to show that NB-WAC can be solved in this model to ensure liveness of the algorithm until the stable period begins.

5.2.1 Solving NB-WAC

[ACT98] presents an algorithm to solve uniform consensus in the crash-recovery model using one of their failure detectors $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}_u$. We can show that NB-WAC can be solved in this model if there are no link failures. To do that, we adapt the crash-model based transformation presented in [Gue95]. See Figure 5.1 for the algorithm.

```
function NB−WAC($vote_i$)
    send ($vote_i$) to all
    d ← $\mathcal{D}_i$
    for all nodes $p_j$
        wait until
                (1) received ($vote_j$) from $p_j$ or
                (2) $p_j \notin \mathcal{D}_i.trustlist$ or
                (3) $d.epoch[p_j] < \mathcal{D}_i.epoch[p_j]$ or
                (4) received (RECOVERED) from $p_j$
        if (2) or (3) or (4) or $vote_j$ = VOTE_ABORT
            return uniformConsensus(ABORT)
    return uniformConsensus(COMMIT)

upon recovery:
    send (RECOVERED) to all
```

Figure 5.1: *Solving NB-WAC in the crash-recovery model*

$\mathcal{D}_i$ represents the failure detector of class $\diamond\mathcal{S}_e$ at processor i, which outputs a trust list and an epoch number for every process in the trust list. $\diamond\mathcal{S}_e$ satisfies the following properties specified in [ACT98]:

- *Monotonicity*: At every good process, eventually the epoch numbers are nondecreasing.

- *Completeness*: For every bad process b and for every good process g, either eventually g permanently suspects b or b's epoch number at g is unbounded.

- *Accuracy*: For some good process K and for every good process g, eventually g permanently trusts K and K's epoch number at g stops changing.

We can now show that the above algorithm solves NB-WAC.

Theorem 9. *The above algorithm reduces NB-WAC to uniform consensus in the crash-recovery model without link failures with either $\diamond\mathcal{S}_e$ or $\diamond\mathcal{S}_u$.*

Proof. As the only difference between $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}_u$ is a strictly stronger accuracy property in $\diamond\mathcal{S}_u$, it suffices to show the proof for $\diamond\mathcal{S}_e$.

- *Termination: Every correct participant eventually decides.* Once the algorithm has reached one of the *return* lines, termination follows from the termination property of uniform consensus. Thus, we only have to show that the while loop eventually terminates. If p_j is good, either the vote (if p_j is up) or RECOVERED (if p_j has crashed and recovered) is received. If p_j is bad, the completeness property of the failure detector guarantees that either condition (2) or (3) is satisfied.

- *Integrity: A participant decides at most once.* As a participant decides by leaving the function and the function is entered only once, integrity holds.

- *Uniform Agreement: No two participants decide differently.* Follows directly from the uniform agreement property of uniform consensus.

- *Validity: If a participant decides* COMMIT *then all participants have voted* VOTE_COMMIT*.* Assume by contradiction that some participant p decides COMMIT although some participant q has voted VOTE_ABORT. If p decides COMMIT, the consensus decision must have been COMMIT. Thus, some process r must have used COMMIT as the input for consensus, i.e. it must have executed line 12. This line, however, can only be reached by r if it received a vote VOTE_COMMIT from every node, including q. A contradiction.

- *Non-Triviality:* Because of the different type of failure recognition in the crash-recovery model, the non-triviality condition can be defined as follows: *If all participants vote* VOTE_COMMIT*, no participant crashes, there is no failure suspicion, and no epoch number on any participant increases then the outcome decision is* COMMIT*.* For the outcome decision to be ABORT, some process p must have used ABORT as the input for consensus, i.e. line 11 must have been executed on process p. However, (2) and (3) would imply that at least one process was suspected, (4) can only be satisfied if at least one process crashed, and $vote_j =$ VOTE_ABORT contradicts the assumption that all participants voted VOTE_COMMIT. Thus, the required contradiction is reached.

□

5.2.2 Topology Construction Algorithm

Note that because of the accuracy property of the failure detector, the above non-triviality condition is strong enough to guarantee that the right proposals will be accepted during the stable period.

Running multiple instances of NB-WAC in series or concurrently, however, imposes the problems discussed in Section 9 of [ACT98]: Stable storage is required to prevent "old" RECOVERED messages to interrupt a "new" NB-WAC instance. Note that stable storage is only necessary to store (1) information about which NB-WAC instances the node participates in at the moment so that it can tag the RECOVERED message appropriately and (2) to store the proposal and decision values used internally by the uniform consensus protocol. As the uniform consensus algorithms presented in [ACT98] also face this problem, this is not an additional restriction imposed by NB-WAC.

As group consistency is checked regularly, outdated state is not a problem, with one exception: It is possible that a node misses the finalizing event of a group creation by crashing and recovering. Thus, if stable storage is also used for the topology construction algorithm, and, thus, the node still has group information from before the crash, *LockedBy* must be reset for all groups to ensure that consistency checking is performed for them:

```
upon recovery:
    for all $gid$ in $Group$
        $Group[gid].LockedBy \leftarrow \perp$
```

Note that the invariants proven in Section 4.1, which are the basis for the covergence proof, still hold in the recovered state.

From this result and the results in [ACT98] we can conclude that our tolopogy construction algorithm can be implemented in the crash-recovery model using $\diamond\mathcal{S}_e$ or a stronger failure detector and stable storage, as long as the number of always-up nodes is greater than the number of bad nodes (see Section 5 in [ACT98] for details about this requirement). Note that this requirement refers to *nodes participating in one particular consensus/NB-WAC instance*, not to *nodes participating in the topology construction algorithm*. As the algorithm can require an arbitrary Δ-sized subset of Π to participate in NB-WAC, this actually requires more always-up than bad nodes in every Δ-sized subset of Π. Therefore, the silent participant extension from Section 3.6.2 would be required to reduce this requirement to the weaker requirement of having more always-up than bad nodes within Π.

5.3 Link Failures

So far only reliable links were considered. However, link failures are likely to occur in real systems. It is easy to see that arbitrary link failures could lead to permanent network partitioning, rendering the problem of creating the unique overlay graph unsolvable. Two weaker models of link failures presented in [BCBT96] include *eventually reliable* (ER) links, where there is a time after which all messages sent are eventually received, and *fair lossy* (FL) links, which guarantee that if an infinite number of messages are sent, an infinite subset of these messages is received.

5.3.1 Crash Failure Model

The following relationships between these models in the presence of crash failures have been identified: Trivially, an eventually reliable link satisfies the properties of a fair lossy link. [BCBT96] has shown that reliable links are “strictly stronger” than eventually reliable links, i.e. there are problems that can be solved with reliable links but cannot be solved in the presence of eventually reliable links. They also show that some problems solvable with reliable links remain solvable with fair lossy links but become unsolvable with unreliable links. Thus, the following relation holds:

$$\text{reliable} \subset \text{ER} \subseteq \text{FL} \subset \text{unreliable}$$

Another result is that, if a majority of nodes is correct, fair lossy links can simulate reliable links. Therefore, if this condition is satisfied, NB-WAC can be solved and the topology construction algorithm works as specified.

5.3.2 Crash-Recovery Model

In the crash-recovery model, [ACT98] provides us with a uniform consensus algorithm tolerating fair lossy links. We can adapt the transformation from NB-WAC to consensus in Section 5.2 by replacing *send* in lines 2 and 15 with *s-send* and defining the following function:

```
function s−send(message) to $p_j$
    fork new thread:
        loop                /* repeat forever */
            send message to $p_j$
```

Theorem 10. *The modified algorithm reduces NB-WAC to uniform consensus in the crash-recovery model with fair lossy links and $\diamond\mathcal{S}_e$ or $\diamond\mathcal{S}_u$.*

Proof. Again, due to the reducability from $\diamond\mathcal{S}_u$ to $\diamond\mathcal{S}_e$ it suffices to show the proof only for $\diamond\mathcal{S}_e$. The proofs for *Integrity*, *Uniform Agreement*, *Validity* and *Non-Triviality* are completely analogous to the proof of Theorem 9. Note that, if some node p_j is good, *s-send* guarantees that either the vote or the RECOVERED message eventually arrives at all good nodes, because only a finite subset of the infinite number of sent messages can be lost. Thus, the *Termination* proof from Theorem 9 also applies. □

6 Conclusion

6.1 Further Work

Although this work contains a complete correctness and implementability proof, some question have been left open and might be worth of further investigation:

- This work solves NB-WAC by reducing it to consensus. NB-WAC, however, might be solvable with a weaker failure detector than consensus, thus reducing the requirements regarding failure detectors or a certain percentage of correct nodes for the topology construction algorithm. Further research regarding the weakest failure detector for solving NB-WAC will be necessary to answer this question.

- In the crash-recovery model, unstable nodes can disrupt large parts of the topology if they are on a low level within the topology tree. An approach where nodes are "punished" for being unstable by giving them penalty weight and thus moving them up the topology tree might be worth investigating, especially as epoch numbers already provide this kind of information. Positioning those nodes on a higher level means that a crash would disrupt less other groups and therefore improve failure locality.

- Chapter 5 partly references costly simulation-based proofs to show that the algorithm can also be implemented in more general system models. These solutions might not be the most efficient; thus, further analysis could yield better solutions.

- As shown in Chapter 5, NB-WAC requires a majority of correct processes to terminate under certain system model conditions. However, it might be possible that (1) the safety conditions (agreement etc.) are still satisfied even if a majority of correct processes fails and (2) the topology construction algorithm can be modified such that termination is not necessary for a NB-WAC instance in which node crashes occur. This option would be worth investigating as it could be used to drop the $f < \lceil \frac{\Delta}{2} \rceil$ requirement even with only an eventually perfect failure detector.

- The algorithm presented in Section 5.3.2 for solving NB-WAC in the crash-recovery model in the presence of fair lossy links is not quiescent, i.e. it does not eventually stop sending messages. [ACT00] analyzes the problem of quiescent reliable communication in the presence of crash and link failures. Generalizing their results to the crash-recovery model would allow designing a quiescent algorithm solving NB-WAC in the crash-recovery model in the presence of lossy links (or proving its impossibility).

6.2 Results

After giving an introduction to the topology construction method developed by Thallner et al. [TS04], a pseudo code implementation of the distributed algorithm, a detailed description and its correctness proof were presented. The algorithm can use arbitrary propose modules [Tha04a]; with a perfect propose module it will construct the unique minimal overlay graph if the network stays stable long enough. The only communication primitive used is non-blocking weak atomic commitment, an agreement problem which has been shown to be reducible to uniform consensus (see [Gue95] for the crash model and Section 5.2.1 of this work for the crash-recovery model). The algorithm does not lose logical safety and liveness during crash failures and can also cope with recovering nodes, either with an empty state and a new node id, or an old state and the original id. Given an appropriate number of correct nodes, even fair lossy or eventually reliable links can be tolerated.

As a by-product, this thesis also contains a reduction from non-blocking weak atomic commitment to uniform consensus in the crash-recovery model.

Bibliography

[ACT98] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *International Symposium on Distributed Computing*, pages 231–245, 1998.

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On quiescent reliable communication. *SIAM J. Comput.*, 29(6):2040–2073, 2000.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, 2004.

[BCBT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Solving problems in the presence of process crashes and lossy links. Technical report, 1996.

[BT93] Özalp Babaoglu and Sam Toueg. Non-blocking atomic commitment. pages 147–168, 1993.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DGFG$^+$04] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346. ACM Press, 2004.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gue95] R. Guerraoui. Revisiting the relationship between non blocking atomic commitment and consensus problems. In *Distributed Algorithms (WDAG-9).* Springer Verlag (LNCS), September 1995.

[Gue02] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distrib. Comput.*, 15(1):17–25, 2002.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. pages 97–145, 1993.

[MT04] Heinrich Moser and Bernd Thallner. Distributed construction of fault-tolerant overlay networks: Construction algorithm. Research Report 39/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, 2004. http://www.ecs.tuwien.ac.at/W2F/RR39-2004.pdf.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Ray97] Michel Raynal. A case study of agreement problems in distributed systems: non-blocking atomic commitment. In *Proceedings of High-Assurance Systems Engineering Workshop*, pages 209–214, Aug 1997.

[Tha04a] Bernd Thallner. Distributed construction of fault-tolerant overlay networks: Propose modules. Research Report 38/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, 2004. http://www.ecs.tuwien.ac.at/W2F/RR38-2004.pdf.

[Tha04b] Bernd Thallner. Fault tolerant communication topologies for wireless ad hoc networks. In *1st Workshop on Dependability Issues in Wireless Ad Hoc Networks and Sensor Networks (DIWANS'04)*, Florence, Italy, June 2004. http://www.ecs.tuwien.ac.at/W2F/documents/diwans04.pdf.

[TS92] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.

[TS04] Bernd Thallner and Ulrich Schmid. Distributed construction of fault-tolerant overlay networks. Research Report 35/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, 2004. http://www.ecs.tuwien.ac.at/W2F/RR35-2004.pdf.

[wp:04] Automata theory (10 dec 2004, 10:10 utc). In *Wikipedia: The Free Encyclopedia*. 2004.