Distributed Construction of Fault-Tolerant Overlay Networks: Construction Algorithm

Heinrich Moser and Bernd Thallner
Technische Universität Wien
Embedded Computing Systems Group E182/2
Treitlstrasse 3, A-1040 Vienna, Austria
Email: {moser, thallner}@ecs.tuwien.ac.at
Tel: +43-1-58801-18203, FAX: +43-1-58801-18297

Abstract

Thallner et. al. [3] 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.

*This research is part of the W2F-project (http://www.auto.tuwien.ac.at/Projects/W2F/), which has been supported by the Austrian START program Y41-MAT.
1 Construction Algorithm

This section presents the complete distributed algorithm of Section 4 in [3] and the missing correctness proofs.

1.1 Introduction

The algorithm requires the following:

- a propose module on each node,
- a non-blocking atomic commit service,
- a fully connected asynchronous system,
- two primitives make connection and cancel connection that create and remove connections in the overlay graph.

The algorithm then guarantees that:

- A consistent, Δ-connected overlay graph is produced.
- The overlay graph is eventually stable, i.e. if there are no changes to 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.
- The algorithm tolerates and adapts to node failures (crash failures).

1.2 Prerequisites

We only guarantee the correctness of the algorithm in case of crash failures. Thus, if a crashed node “recovers”, it must re-enter the system as a new node.

We also assume that the propose module does not deliver impossible proposals, for example, 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 paper.

atomic commit refers to an implementation of the non-blocking atomic commit protocol (NBAC, such as the one described in [2]), i.e. it must satisfy the uniform agreement, uniform validity, termination and non-triviality requirements. Informally, atomic commit must only return COMMIT if all participants of the atomic commit process decide on VOTE_COMMIT and no process is suspected by the failure detector. On the initiator, initiate atomic commit is called. As the implementation of this building block is described in-depth in other papers, it is omitted here for brevity.

We provide the functions vote...(), decision...() and finalize...() to parameterize the atomic commit. vote...() is called to locally decide on VOTE_COMMIT or VOTE_ABORT, decision...() is executed on every participating node after the atomic commit algorithm reached a common decision, and finalize...() is run after decision...() has been executed on all participating nodes.\(^1\)

Note that additional constraints (failure detectors, reliable links, etc.) might be imposed on the system by the requirement for a non-blocking atomic commitment protocol.

Our algorithm does not use a failure detector directly; however, we require the atomic commit algorithm to behave as if provided with an eventually perfect failure detector as specified in [1]. Thus, up to a certain point in

\(^1\)Note that this finalizing atomic commit can be easily implemented as a combination of two (not early deciding) atomic commit sequences, where decision_1() unblocks vote_2() and decision_2() is used as the overall finalize().
time, the atomic commit algorithm is allowed to make false suspicions and decide ABORT although all participants voted COMMIT. However, the atomic commit is not allowed to decide COMMIT if one participant crashed (and, therefore, did not vote at all). Formally, it must satisfy the following requirements:

- **Strong Completeness.** No atomic commit may decide on COMMIT if a crashed process participates.
- **Eventual Strong Accuracy.** There is a time after which all atomic commits decide on COMMIT if all participants vote VOTE, COMMIT and no crashed process participates.

Chandra and Toueg also mention in [1] that in actual systems these requirements can be satisfied long enough for an algorithm to do its work using a simple timeout-based failure detector.

### 1.3 Data Structures

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 3 in [3] for details).

A proposal, as defined in Section 4 in [3], 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 in [3].
- **Connections:** A set of \( \frac{\Delta(\Delta-1)}{2} \) connection structures. A connection structure must contain a set of two endpoints and a positive cost or weight of the connection. Other information needed by `make connection`, `cancel connection` or the propose module can 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.
- **Finished:** True, 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).

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.\(^2\) 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.

The main data structure used by the algorithm is the *Group* associative array. It uses general group ids (i.e. sets of one or Δ node ids) as keys; the values are group structures. An example representing the overlay graph of Figure 1 in [3] can be found in Figure 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 is that a 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 a terminal node \( p \) of a group \( X \), which is not leader of \( X \), might

\(^2\)Actually, the value of Terminals does not matter for nodes stored in group structures. It is, however, useful to have \( gid = Group[gid],Terminals \) to be consistent with Section 3 in [3].
become leader of a higher level group $Y$, with $X$ being a member of $Y$. Thus, $p$ must know about $Y$ (1) to determine whether it is leader of $Y$ and, if it is, (2) to 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 $Group[ gid ] \neq \perp$. Thus, we decided to drop the intuitive requirement for a leader and have all terminal nodes of all members of a group make decisions for that group. See Figure 2 for an example based on Figure 1 in [3].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>{}</td>
<td>(A){1, 2, 3}</td>
</tr>
<tr>
<td>{2}</td>
<td>{}</td>
<td>(A){1, 2, 3}</td>
</tr>
<tr>
<td>{3}</td>
<td>{}</td>
<td>(A){1, 2, 3}</td>
</tr>
<tr>
<td>{4}</td>
<td>{}</td>
<td>(B){4, 5, 6}</td>
</tr>
<tr>
<td>{5}</td>
<td>{}</td>
<td>(B){4, 5, 6}</td>
</tr>
<tr>
<td>{6}</td>
<td>{}</td>
<td>(B){4, 5, 6}</td>
</tr>
<tr>
<td>{7}</td>
<td>{}</td>
<td>(C){1, 6, 7}</td>
</tr>
<tr>
<td>{8}</td>
<td>{}</td>
<td>(D){8, 9, 10}</td>
</tr>
<tr>
<td>{9}</td>
<td>{}</td>
<td>(D){8, 9, 10}</td>
</tr>
<tr>
<td>{10}</td>
<td>{}</td>
<td>(D){8, 9, 10}</td>
</tr>
<tr>
<td>{11}</td>
<td>{}</td>
<td>(F){11, 12, 13}</td>
</tr>
<tr>
<td>{12}</td>
<td>{}</td>
<td>(E){1, 10, 12}</td>
</tr>
<tr>
<td>{13}</td>
<td>{}</td>
<td>(F){11, 12, 13}</td>
</tr>
</tbody>
</table>

| A                                      | \{1, 2, 3\}                      | (C)\{1, 6, 7\}                |
| B                                      | \{4, 5, 6\}                       | (C)\{1, 6, 7\}                |
| C                                      | \{1, 6, 7\}                       | (C)\{1, 6, 7\}                |
| D                                      | \{8, 9, 10\}                      | (E)\{1, 10, 12\}              |
| E                                      | \{1, 10, 12\}                     | (E)\{1, 10, 12\}              |
| F                                      | \{11, 12, 13\}                    | (F)\{11, 12, 13\}             |

Figure 1. Group map for Figure 1 in [3]

1.4 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\(^3\) and is used to detect changing connection weights (e.g. because of moving nodes).

1.5 Description

The algorithm consists of single threaded, message-driven code.

\(^3\)For reasons of simplicity we assume that this value is known by the endpoint with the lowest node id.


<table>
<thead>
<tr>
<th>Node</th>
<th>Group[...] is set on this node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}, {(A){1,2,3}, (C){1,6,7}, (E){1,10,12}, (F){11,12,13}}</td>
</tr>
<tr>
<td>2</td>
<td>{2}, {(A){1,2,3}, (C){1,6,7}}</td>
</tr>
<tr>
<td>3</td>
<td>{3}, {(A){1,2,3}, (C){1,6,7}}</td>
</tr>
<tr>
<td>4</td>
<td>{4}, {(B){4,5,6}, (C){1,6,7}}</td>
</tr>
<tr>
<td>5</td>
<td>{5}, {(B){4,5,6}, (C){1,6,7}}</td>
</tr>
<tr>
<td>6</td>
<td>{6}, {(B){4,5,6}, (C){1,6,7}, (E){1,10,12}}</td>
</tr>
<tr>
<td>7</td>
<td>{7}, {(C){1,6,7}, (E){1,10,12}}</td>
</tr>
<tr>
<td>8</td>
<td>{8}, {(D){8,9,10}, (E){1,10,12}}</td>
</tr>
<tr>
<td>9</td>
<td>{9}, {(D){8,9,10}, (E){1,10,12}}</td>
</tr>
<tr>
<td>10</td>
<td>{10}, {(D){8,9,10}, (E){1,10,12}, (F){11,12,13}}</td>
</tr>
<tr>
<td>11</td>
<td>{11}, {(F){11,12,13}}</td>
</tr>
<tr>
<td>12</td>
<td>{12}, {(E){1,10,12}, (F){11,12,13}}</td>
</tr>
<tr>
<td>13</td>
<td>{13}, {(F){11,12,13}}</td>
</tr>
</tbody>
</table>

Figure 2. Who knows what in Figure 1 in [3]

1.5.1 Main loop

```plaintext
var Group[]

Group[ID].Members = {}
Group[ID].Weight = 0
Group[ID].Connections = {}
Group[ID].Terminals = [ID]
Group[ID].ParentID = ⊥
Group[ID].Finished = true

loop
    wait for incoming message
    if received proposal P from local proposal module
        initiate atomic commit PROPOSE_GROUP:
            participants = all terminal nodes of all P.Members
            data = P
    if received signal to check for broken groups
        var gid = Group[ID].ParentID
        while gid ≠ ⊥ ∧ Group[gid].Finished = true
            initiate atomic commit CHECK_GROUP:
                participants = all terminal nodes of all Group[gid].Members
                data = Group[gid]
                gid = Group[gid].ParentID
    if received atomic commit message
        execute atomic commit phase
```

The group structure is initialized only with the single-node group of the node itself. New proposals created by the proposal module of the node are distributed to the terminal nodes of all members of the proposed group. An atomic commit 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 check 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 \( y \) left. The periodic group consistency check on each node ensures that such changes are detected and updated in the local data structures. As a side-effect, this check also detects node crashes and changed weights of groups (see below). The check starts directly above the node itself (line 16) and then travels “upwards” the topology graph (line 21, see Figure 1(a) in [3]). Note that detecting node crashes and changed group weights is necessary anyway, which is why we chose to also detect “left” groups using this method rather than using a (possibly more efficient) notification via traditional message passing.

As a non-blocking atomic commit is usually implemented as a multi-phase protocol, we must listen for messages starting the next phase of an ongoing atomic commit and execute the corresponding phase. Note that, on each node, many atomic commit 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 commit sequence in line 23 (or initiate a new sequence, if an initial atomic commit message arrives).

Note that both atomic commit variants (\texttt{PROPOSE\_GROUP} and \texttt{CHECK\_GROUP}) use the set of all terminal nodes of all members of a group as participants. Thus, \( \Delta \) (all members are single-node groups) to \( \Delta^2 \) (all members are composite groups) nodes participate in each atomic commit. This is used to distribute the information about the join or leave to all these nodes. See Section 1.3 for an explanation as to why this is necessary.

1.5.2 Atomic commit functions

\begin{verbatim}
24 function vote\_PROPOSE\_GROUP(group) /* local decision of atomic commit */
25 var gid—element gid of group.Members with ID \( \in \) gid
26 if want\_to\_join(gid, group)
27    return VOTE\_COMMIT /* better group found */
28 else
29    return VOTE\_ABORT /* current group is better */
30
31 procedure decision\_PROPOSE\_GROUP(result, group) /* locally execute atomic commit decision */
32 var gid—element gid of group.Members with ID \( \in \) gid
33 if result = COMMIT
34    if want\_to\_join(gid, group) /* check again, something might have changed */
35       join\_group (gid, group)
36
37 procedure finalize\_PROPOSE\_GROUP(result, finalize\_result, group) /* decision has been executed everywhere */
38 var gid—element gid of group.Members with ID \( \in \) gid
39 if is\_locally\_consistent (group) \&\& result = COMMIT /* see below */
40 if finalize\_result = ABORT /* node crash after decision */
41 leave\_group (gid)
42 if finalize\_result = COMMIT
43   Group[group.Terminals].Finished—true
44
45 function want\_to\_join (gid, group) /* true , if member gid should join group */
46 return \\
47   (Group[gid].ParentID = \bot \lor group.Weight < Group[Group[gid].ParentID].Weight) \& \& \\
48  (Group[group.Terminals] = \bot \lor Group[group.Terminals].Weight > Group[gid].Weight) \& \& \\
49  group.Weight > Group[gid].Weight

\end{verbatim}
The atomic commit algorithm 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`.

**PROPOSE_GROUP**  The nodes only decide on `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 commit 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 in [3]).

**Finished** is set 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 commit 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 in [3] (implemented as function `calculate_weight()`).

### 1.5.3 Joining and leaving groups

#### procedure `join_group(gid, group)` /* member gid joins new group */

- **if** `Group[gid].ParentID ≠ ⊥`
- **leave_group(gid)`
- `Group[group.Terminals] ← group`
- **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 ≠ ⊥`
- **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 ← ⊥`

Both procedures assume that `Group[gid] ≠ ⊥`, which is checked in `decision...()`.

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

*leave_group()* removes the connections built by the group and removes unnecessary data structures. It requires that `Group[Group[gid].ParentID] ≠ ⊥`. The calling procedures (*join_group(), decision...*) only guarantee that `Group[gid].ParentID ≠ ⊥`. However, Theorem 1 will prove that those two assertions are equal.

### 1.6 Correctness

A **stable period** is a period in time during which no nodes crash, 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''` (\(Π = Π' \cup Π''\) and \(n = n' + n''\)). According to Theorem 2 in [3], there is a unique overlay graph `G'` which our algorithm is supposed to create. As mentioned in the theorem, we require that `n' ≥ 1` and `n'' ≥ 2Δ - 2`. Note that if these requirements are not fulfilled, the algorithm does not lose its 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. See Section 3 in [3] for an extensive definition of the difference between regular and gateway nodes.

To show that the algorithm is correct, we prove 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.
First, we need some definitions ($p$ being a node id):

- $\text{Group}_p$ is the local variable $\text{Group}$ at node $p$.
- $\text{Parent}^i_p$ is the “$i$-th parent of $p$”. Formally, it is recursively defined as follows:
  
  - $\text{Parent}^0_p := p$
  - $\text{Parent}^n_p := \text{Group}_p[\text{Parent}^{n-1}_p].\text{ParentID}$

- $\text{Parentlist}_p$ is defined as the list $(\text{Parent}^0_p, \text{Parent}^1_p, \ldots, \text{Parent}^n_p)$, with $\text{Parent}^n_p \neq \bot$ and $\text{Parent}^{n+1}_p = \bot$.

### 1.6.1 Consistency

**Lemma 1.** The assignment

```plaintext
var gid ← element gid of group.Members with ID ∈ gid always returns exactly one value for gid.
```

**Proof.** The assignment occurs at the beginning of atomic commit functions. $\text{group}$ is passed as a parameter of the atomic commit. Thus, node $ID$ participates in an atomic commit with $\text{group}$ 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 $\text{group}$’s members. However, lines 12 and 18 show that all atomic commit participants are terminal nodes of at least one member of $\text{group}$. A contradiction.

We now assume that more than one value for $gid$ is returned. This contradicts our assumption in Section 1.2 that no impossible proposals (i.e. proposals where two members share a common terminal node) are created. As the $\text{Member}$ field of group structures is never changed, this property also holds for the group structure passed to the CHECK\_GROUP atomic commit.

**Theorem 1.** For all nodes $p$: At the beginning of each iteration of the main loop (i.e. every time the execution reaches line 10), the following invariants hold ($m, g \neq \bot$):

1. The node only knows about all groups in its parent list.
   \[
   \forall g : g \in \text{Parentlist}_p \Rightarrow \text{Group}_p[g] \neq \bot
   \]

2. $p$ is a terminal node in all parent list entries except for the last one.
   \[
   \forall i, 0 \leq i \leq |\text{Parentlist}_p| - 2 : p \in \text{Group}_p[\text{Parent}^i_p].\text{Terminals}.
   \]

3. The weights of the parent list entries are strictly increasing.
   \[
   \forall m, g : \text{Group}_p[m].\text{ParentID} = g \Rightarrow \text{Group}_p[g].\text{Weight} > \text{Group}_p[m].\text{Weight}
   \]

4. The parent list is finite.
   \[
   \exists i : \text{Parent}^i_p = \bot
   \]

5. The parent-member relationship is consistent.
   \[
   \forall m, g : \text{Group}_p[m].\text{ParentID} = g \Leftrightarrow m \in \text{Group}_p[g].\text{Members} \land p \in m
   \]

Note that, as $\text{Finished}$, $\text{ParentID}$ and $\text{Weight}$ are the only fields that are changed in the algorithm, $\text{Group}_p[g] \neq \bot$ implies that all other fields of $\text{Group}_p[g]$ contain all the information initially provided by the proposal that was assigned to $\text{Group}_p[g]$ in $\text{join\_group}()$. 

9
Proof. Execution of the algorithm on a node can be seen as a series of computation events, each corresponding to an iteration of the main loop (until line 10 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 1.5.1 shows that, initially,

- \( \text{Group}_p[p] \neq \bot \)
- \( \text{Group}_p[p].\text{ParentID} = \bot \), and thus, \( \text{Parentlist}_p = (p) \)
- \( \text{Group}_p[p].\text{Members} = \{\} \)
- \( \forall q \neq p : \text{Group}_p[q] = \bot \)

Thus, initially, the five invariants hold.

For the induction step, we analyse the following code blocks in which the \( \text{Group} \) array is changed and which could theoretically violate the invariants. Note that we can ignore changes to \( \text{Parentlist}_p \) or \( \text{ParentID} \) of the last element in the parent list, equals \( \bot \), thereby also satisfying \( \bot \) as the third invariant proves that there are no duplicate entries in the parent list) and therefore still satisfies the fifth invariant. \( \text{Group} \) subroutines they call. By showing that these functions do not violate the invariants, we prove the correctness of the theorem.

\[ \text{check_group} \] First we show that the parameter passed to \( \text{leave_group} \) is in \( \text{Parentlist}_p \). If result = \( \text{ABORT} \), line 69 guarantees that \( \text{leave_group} \)'s parameter \( \text{gid} \) satisfies \( \text{Group}[\text{gid}] \neq \bot \). Thus, as we can assume that the invariants hold for all previous configurations, \( \text{gid} \in \text{Parentlist}_p \). If result = \( \text{COMMIT} \), the call to \( \text{is_locally_consistent} \) guarantees that \( \text{Group}[\text{group}\text{.Terminals}] \neq \bot \) (and, thus, \( \text{group}\text{.Terminals} \in \text{Parentlist}_p \)). Additionally, Lemma 1 guarantees that \( \text{gid} \) contains a valid value which, by definition, is in \( \text{group}\text{.Members} \). We know that the fifth and the first invariant hold in the previous configuration and, therefore, \( \text{Group}[\text{gid}] \neq \bot \) and \( \text{gid} \in \text{Parentlist}_p \). Thus, all parameters passed to \( \text{leave_group}() \) contain parameters in \( \text{Parentlist}_p \). To work correctly, \( \text{leave_group}() \) also requires that \( \text{Group}[\text{Group}[\text{gid}].\text{ParentID}] \neq \bot \), which is, however, satisfied at all code lines where \( \text{leave_group}() \) is called.

The loop in \( \text{leave_group} \) sets the group structures of all entries in \( \text{Parentlist}_p \) following (but not including) \( \text{gid} \) to \( \bot \). \( \text{Group}[\text{gid}].\text{ParentID} \) is also set to \( \bot \), making it the last entry in \( \text{Parentlist}_p \). Thus, the first invariant holds after execution of \( \text{leave_group}() \).

As no new entries are added to \( \text{Parentlist}_p \), the second and forth invariants are still satisfied. If no group weights are changed (line 62), the same holds for the third invariant. Otherwise, assume that the third invariant is violated in line 62. There are two possibilities: \( \text{Group}[\text{group}\text{.Terminals}]\text{.Weight} \) can become smaller or equal to weight of the previous parent list entry (\( \text{gid} \)). This is checked in line 63 and corrected in the following line by removing all entries following \( \text{gid} \) from the parent list. The other case, \( \text{Group}[\text{group}\text{.Terminals}]\text{.Weight} \) becoming greater or equal to its parent, is checked in line 65 and corrected in the following line by removing all entries following \( \text{group}\text{.Terminals} \) from the parent list.

To prove the fifth invariant it suffices to look at the group structures contained in \( \text{Parentlist}_p \) as the first invariant shows that these entries are the only ones having \( \text{ParentID} \) or \( \text{Members} \) set. Let \( i \) be the index of \( \text{leave_group} \)'s parameter \( \text{gid} \) in the parent list. The group structures of entries \( \text{Parent}^0_p \) to \( \text{Parent}^{i-1}_p \) 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. \( \text{Group}[\text{Parent}^0_p].\text{ParentID} = \text{Parent}^0_p, \text{Parent}^{i-1}_p = \text{Group}[\text{Parent}^i_p].\text{Members} \) and \( \text{ID} \in \text{Parent}^{i-1}_p \) also hold because those values have not been changed. \( \text{Group}[\text{Parent}^0_p].\text{ParentID} \), the \( \text{ParentID} \) of the last element in the parent list, equals \( \bot \), 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] ≠ ⊥` and `Group[Group[gid].ParentID] ≠ ⊥`. `join_group`’s parameter `gid` fulfills the first requirement, which is guaranteed in function `want_to_join()`. The second requirement is checked in line 82 (using the first invariant). Thus, the call to `leave_group()` in line 83 returns with a consistent group structure. Note that after line 83, `gid` is the last entry in the parent list, because either line 82 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 ⊥ (thus ending the parent list), because proposals always have `ParentID = ⊥`. This is consistent with the first invariant. The second invariant holds because of the definition of `gid` in line 32 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] ≠ ⊥`) and, thus, the group entry gets overwritten in line 84. 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 ∈ (Parent<sub>0</sub>ᵢ, …, Parent<sub>i</sub>ᵢ = gid)`. We know that `want_to_join()` evaluated to true, that `Group[Group.Terminals] ≠ ⊥` and, therefore, `Group[Group.Terminals].Weight > Group[gid].Weight` (see line 48). 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 ∈ (Parent<sub>ᵢ⁺¹</sub>ᵢ, …, Parent<sub>|Parentlist|⁻¹</sub>ᵢ)`. 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 84 `Group[group.Terminals].ParentID = ⊥` and, therefore, does not need to be considered. Thus, we only have to show that after `join_group()` `gid ∈ Group[group.Terminals].Members` and `p ∈ gid`. Note that `Group[group.Terminals] = group`. The definition of `gid` in line 32 thus guarantees that the fifth invariant is not violated.

\[\square\]

1.6.2 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. Note that there are no direct or indirect recursions in the algorithm. There are two loops that need to be analyzed:

First, there is the the **do-while** loop in `leave_group()`. We have shown in the proof of Theorem 1 that, when `leave_group()` is called, `Group[gid] ≠ ⊥`. Thus, according to Theorem 1, `gid ∈ Parentlist_p`. The loop traverses the parent list until it reaches the end (`del.id = ⊥`). 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 17. 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.

Thus, all iterations of the main loop eventually terminate, and all messages in the message queue will eventually be received in line 10.

Lemma 3. All atomic commit sequences eventually terminate.

Proof. As the atomic commit 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.

1.6.3 Convergence

We use the following definitions:

- An active commit sequence is finished iff finalize\_\... () (or decision\_\... (), if finalize\_\... () does not exist), has been executed on all participating nodes.

- An active commit sequence is active, iff it has been initiated, but it is not finished yet.

Note that neither a group id nor a group structure can uniquely identify a “group” in the topological meaning. A group structure only stores information about the member ids of the group, not about the composition of these members. Thus, it is possible that two group structures 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 commit 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.

Let \( C = (g_1, \ldots, g_m) \) be the list of all general groups in \( G' \), the unique overlay graph as defined in Theorem 2 in [3], in 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 1. A group structure 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, if, in the current configuration and every configuration until the end of the stable period, all group structures 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.

Lemma 4. For all \( i, n \leq i \leq |C| \) 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.

We now show 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 \( (g_0, \ldots, g_{i-1}) = C_{i-1} \) exist and their group ids are \( i - 1 \)-stable, these ids are by
definition $i$-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.

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 can now show that there is a time $t_2 > t_1$ after which all group structures 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 structure. 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:

**Definition 2.** The level$^{m}_{p}[gid]$ 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 $1 + \max_{p' \in P} \text{level}^{m'}_{p'}[m_{p'}]$, 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 we will use, is to redefine the level as follows:

**Definition 3.** The level$^{m}_{p}[gid]$ of a group id $gid$ on a node $p$ at time $t$ is defined as 0 for single-node group structures. For a composite group structure gid,

- the level is set initially during the decision phase of PROPOSE\_GROUP for a proposal from some proposal module $X$ to $1 + \max_{m \in \text{Group}_{p}[gid.\text{Members}]} \text{level}^{m'}_{p_m}[m]$ 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 $1 + \max_{p' \in P} \text{level}^{p'}_{p}[m_{p'}]$, 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 structure is the level the propose module “thought” that the new group structure 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 structures whose level is less than or equal to level $j$ are i-weighted.
Proof. Proof by induction. Level 0 group structures (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 \textsc{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 existed at \(t_2\) a \textsc{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 structure \(g\) with group id \(gid\) on node \(p\) has a level less than or equal to \(j\) but is not \(i\)-weighted. If the group structure has a level less than \(j\), this contradicts the assumption that the lemma holds for \(j - 1\). If the group structure has level \(j\), we have the following cases:

- The last update to the level of the group structure \(g\) has been a \textsc{propose\_group}. This \textsc{propose\_group} must have been created after \(t_1\) (otherwise, there would have been a \textsc{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 structure 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 \textsc{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 in [3].)

- The last update to the level of the group structure \(g\) has been a \textsc{check\_group}. This \textsc{check\_group} must have been started after \(t_2\). As the group structure has level \(j\), the group structures 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 structures 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 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 structure must have weight greater than the weight of \(g_i\), thereby becoming \(i\)-weighted. Consider the fact that there is only a finite amount of possible discrete group weights. Note also that for every given tuple \((t, p, gid)\) which has a level \(\text{level}_t[p][gid] > 0\) and a weight \(\text{Group}_p[gid].\text{Weight}\) (at time \(t\)) associated to it, we can use the definition of \text{level} to find a tuple \((t', p', gid')\) with \(\text{level}_{t'}[gid'] = \text{level}_t[p][gid] - 1\) and a weight \(\text{Group}_{p'}[gid'].\text{Weight}\) (at time \(t'\)). 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 \texttt{want\_join()} are satisfied:

- \( \text{Group}_p[\text{gid}] \neq \bot: \text{gid} \in C_{i-1} \) and, by assumption, all groups in \( C_{i-1} \) exist permanently.
- \( (\text{Group}_p[\text{gid}].\text{ParentID} = \bot \lor \text{group.Weight} < \text{Group}_p[\text{Group}_p[\text{gid}].\text{ParentID}].\text{Weight}) \): 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.

- \( \text{group.Weight} > \text{Group}[\text{gid}].\text{Weight} \): \( \text{gid} \in C_{i-1} \) 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 left 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 \( C \).

- \texttt{CHECK\_GROUP} returns \texttt{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.

- \texttt{CHECK\_GROUP} returns \texttt{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. it 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 2.** If the stable period lasts long enough, the constructed topology will be the unique overlay graph.

**References**
