MASTERARBEIT

Simulation and Performance Evaluation of a Topology Control Algorithm in NS2

Ausgeführt am Institut für

Institut für Technische Informatik
Embedded Computing Systems Group
der Technischen Universität Wien

unter Anleitung von O.Prof. Dr. Ulrich Schmid

Betreuende AssistentIn Univ. Ass. Dr. Bettina Weiss

Christian Walter

Mat. Nr: 0225458 Knz: 938
Sachsenplatz, 7/11 1200 Wien

Datum

Unterschrift

Abstract

In an ad hoc wireless network distributed nodes communicate with each other over a wireless medium. Two important problems in wireless ad hoc networks are topology control and routing. Topology control can be defined as the problem of maintaining a spanning communication graph. Routing is the process of moving messages across a network from a source to a destination.

This works extends the topology management algorithm from Thallner [TM05], briefly called TMA, which generates a k-regular and k-connected overlay graph. We start by an in depth explanation of the algorithm and show some enhancements to adapt the algorithm to a real network where some of the assumptions cannot be held any more. Then we continue by defining the necessary components needed for a real world implementation of the Thallner algorithm. Using these components, we show how they can be implemented in the network simulator NS2 [FV06]. Chapter 4 shows our simulation results and will comment on them.

The second part of our work, which is presented in Chapter 5 provides an implementation of the Thallner algorithm in a different network model. While the original model uses an asynchronous model with reliable links, the second network model assumes bounded delays with lossy links. This allows for more efficient implementations because algorithms designed for synchronous models can be used. We will present our proposal for an adapted algorithm and will show how it can be implemented in the network simulator.

The final part of this work shows how further studies like the evaluation of routing algorithms can be performed on top of the simulation framework. This includes an example of a flooding protocol which we developed during our studies, and the DSDV ad hoc routing protocol. We have also included some basic guidelines on how to implement other topology control algorithms in NS2.

4

Contents

1 Introduction

1.1 Topology Control and Routing

Topology control is the problem of computing and maintaining a connected topology among all nodes [Raj02, p60]. A *topology graph* $T' = (V', E')$ in this sense is simply a subset of the *transmission graph* $T = (V, E)$ where $V = V'$ is the set of nodes and $E \subseteq E'$ is the set of connections between them. A transmission graph is an abstraction of the real world network where two nodes are connected if they can communicate with each other. Topology management is a low-level service which is typically built directly into or upon the MAC layer. It therefore provides a different view of the network to higher levels by restricting point-to-point communication and favoring multi-hop communication. The reasons for this are manifold and a lot of different metrics exist to judge on this. Typical metrics are *energy efficiency*, *fault tolerance*[1], *robustness to mobility*, *connectivity degree*, *reduced interference*, *message/time complexity* and a lot more.
Common to almost all algorithms is that they use messages to exchange information with neighbors and then use this information to select the "best" neighbors to talk to from the available ones. Such protocols are called *Neighbor-Based* protocols [San05, p.182]. Other variants are *Direction-Based* protocols which ensure that at least one neighbor is in every cone-angle p [San05, p. 182] of a node. Example for such protocols are CBTC - *Cone-Based Topology Control* [LHB$^+$01]. Finally there are *Location-Based* protocols which typically use GPS receivers to generate their topology. A protocol using location information is LMST - *Local Minimum Spanning Tree* [LHSS05]. To integrate fault tolerance, efforts were made to establish $k-$*node connectivity*, for example kXTC [SKSS04], where there are k node redundant paths between any pair of nodes. Since nodes cannot handle an arbitrarily large number of connections, algorithms that provide a $k-$*regular* topology, that is, which restricts the number of connection of every node to k, are of particular interest. The TMA algorithm investigated in this thesis provides both, $k-$ node redundancy and $k-$regularity [Tha05].

Topology management by itself is not enough because messages have to be transmitted between nodes. If the nodes are not directly connected the message must pass an intermediate node. The process of determining a good path for sending a message in a network is called routing and is one of the classic problems in this research area. It can be seen as the distributed version of the shortest-path problem although the metric might be different. To appropriately route messages within a network every node (or group) must have a unique (and known) address. Whenever a node receives a message it either accepts the message if it is the destination, drops the message, or forwards it to another node. This algorithm continues until the message has eventually reached its destination or it has been dropped. More sophisticated routing protocols support *multiple paths* to a single destination. This can be used to provide better throughput by load balancing or reliability by the means of fault tolerance. Such protocols are referred to as *multipath* routing protocols in contrast to *single path* routing protocols. Routing protocols can be classified into *reactive* routing protocols which obtain

[1] Note that restricting communication always implies that the resulting graph has worse or equal properties with respect to fault tolerance than the transmission graph.

their information on demand, or *proactive* routing protocols which try to keep a current view of the topology. Classic reactive routing protocols for adhoc networks are DSR - *Dynamic Source Routing* [JMH03], [JMB01], AODV - *Adhoc On-Demand Distance Vector* [PR99] and TORA - *Temporally-Ordered Routing Algorithm* [PC97]. Pro-Active routing protocols are DSDV - *Destination-Sequenced Distance-Vector* [PB94], WRP - *Wireless Routing Protocol* [MGLA96] and OLSR - *Optimized Link State Routing Protocol* [JMC+01]. *Flooding* of messages, although not directly a routing protocol, is also an option and is often a simple, fault-tolerant and efficient solution for information dissemination. Examples for these protocols are our own *Source-Sequenced-Flooding* protocol presented in Appendix D and OFP - *Optimal Flooding Protocol for Routing in Ad-hoc Networks* [PDDJ02]. Although we did not directly examine the performance of routing protocols on top of the TMA we have presented some examples in the Appendix on how they can be executed on top of the topology graph. Therefore this serves basically as a starting point for further work.

1.2 Network model and basic algorithm

Our theoretical studies are based on an asynchronous network model with reliable transmission where n nodes are fully connected. A more complete definition of our network model is given later in Section 2.1.1. We will now present the basic idea behind our TMA algorithm. Initially every node has k open connections. If we group k nodes together and fully connect them internally then every node has exactly one connection left. Having k nodes this implies that there are still k connections left. This node can be considered as a "super-node" and treated just like a single node. We will call a node with one external connection left a *terminal node*. Note that within a group we have $(k-1)$ redundant paths. Using this method we can build a hierarchical structure, which always has $(k-1)$ redundancy internally. Unfortunately to get k-redundancy by connecting the top-level terminal nodes we need a particular number of nodes, which is not guaranteed in an arbitrary network. To resolve this we require some additional *gateway nodes* which are put in the network and only used if no valid grouping can be found without them. Therefore we have the total number of nodes n which is the sum of the normal nodes n' and the gateway nodes n''. With this addition the TMA can guarantee to find a topology that is k-connected.

The basic operation principle is a fully distributed algorithm executing locally at every node immediately after startup. A *propose module* periodically tries, and if found proposes, new groups. If the proposed groups are better than the current one, then the proposal is broadcasted to all terminal nodes of all group members and a consensus algorithm is executed which decides if nodes accept or refuse the proposal. If the group is accepted, it is built. Because nodes can crash or may leave groups without notifying other members a periodic consistency check is executed.

1.3 Topology Management Algorithm TMA

This work is based on the results from Thallner [Tha05] and Thallner, Moser [TM05]. The Thallner algorithm, briefly called TMA, builds a k-regular and k-node-connected topology from a fully connected transmission graph. The resulting topology has a low overall transmission power using the minimal number of links (approx. $k \cdot n/2$ links). It requires only local information like the set of neighbors and distance/channel loss and constructs and maintains an topology graph, sometimes also called overlay graph.
Its basic operational principle is very simple. The algorithm builds groups of k members, where k is a network parameter chosen by the network designer to match its degree of redundancy. The *members* of a group are either nodes or again groups, and the members are fully connected internally. This leaves k *terminal nodes* with one connection left to redundantly connect the group to the remainder of the network. For a single node, which can be treated as a special form a group, these are simply the k available connections of the node. For a "real" group the connections originate from the terminal nodes which are also used to uniquely identify a group within a network. Such a identifier is called a *group identifier*, briefly *gid*. A group is defined by its members, its group id, its group internal-connections and its weight, which is derived from the weight of its connections and the weight of its members. This information can be written as a quadruple $(group\ identifier, groupid, connections, weight)$.
For example in Figure 1.1 and Figure 1.2 we can see that the nodes $0, 1, 4$ have formed the group $(\{\{0\},\{1\},\{4\}\},\{0,1,4\},\{(0,1,66),(0,4,59),(1,4,61)\},186)$. The weight of the group in this case is 186 which is simply the sum of its group internal connections.

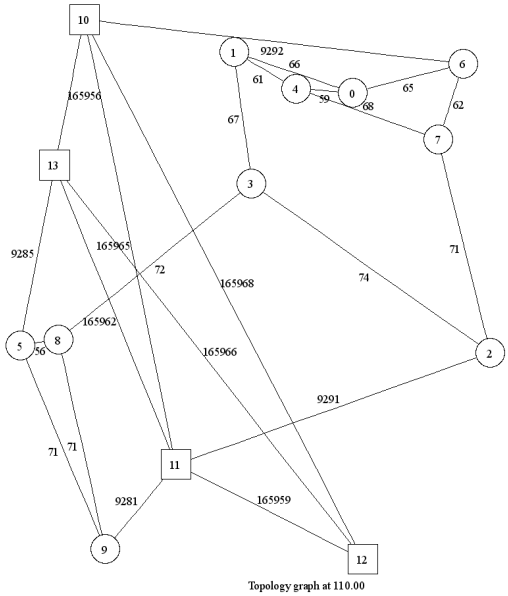


Figure 1.1: Example topology graph created by the TMA for $k = 3$, $n' = 10$ and $n'' = 4$.

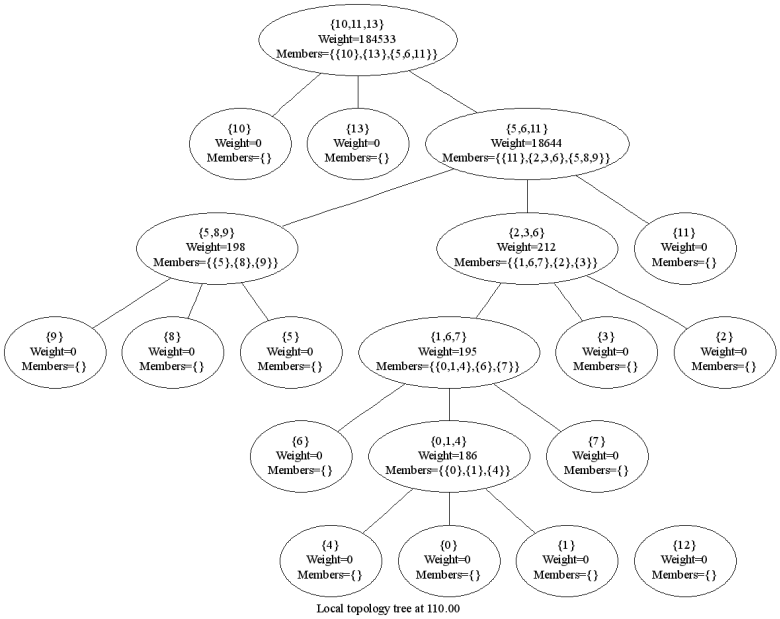


Figure 1.2: Example topology tree created by the TMA for $k = 3$, $n' = 10$ and $n'' = 4$.

We have already mentioned that groups are either built from nodes, like the group $\{0, 1, 4\}$, or members can be groups itself. This can be seen in the group $(\{\{0, 1, 4\}, \{6\}, \{7\}\}, \{1, 6, 7\}, \{(0, 6, 65), (4, 7, 59), (6, 7, 62)\}, 195)$. In this case the terminal nodes 0 and 4 of the group $\{0, 1, 4\}$ are used to build the group internal connections for the group $\{1, 6, 7\}$ and the nodes 1, 6 and 7 still have a connection left after group construction and therefore they become the new terminal nodes. It is also important to note that we can see two different types of nodes in Figure 1.1. The circles are "normal" nodes and the algorithm strives to achieve k-connectivity and k-regularity for them. Since these properties can not be achieved in every network, the algorithm requires additional "gateway nodes", which are shown as boxes, to guarantee these properties in such cases[2].

Now, how are proposals generated? This is the concern of the so-called *Propose Modules*. These modules typically use simple search strategies [TM05, p57] to explore the search space. Different variants where proposed in [TM05, p57-p82] where one of them is covered in depth in Section 2.4. They differ in their level of distribution, if they use local and global information and if they are deterministic or probabilistic. Whenever a propose module releases a proposal an algorithm executes on all terminal nodes of all members of the proposed group (up to k^2) to decide if the members should join the group or not. To ensure the consistency of groups even in presence of crash failures or network changes *periodic group checking* is employed. It adapts the already built groups to the new connection weights and ensures that all groups are consistent among

[2]Note that gateway nodes are only used as a least resort. If it is possible to construct a $k - regular$ topology without them, they are not used.

all nodes. Inconsistent groups are destroyed. Combining all these different facilities the algorithm generates a k-connected and k-regular topology graph which adapts to changing network weights and nodes.

In his thesis, Thallner has covered all the theoretical aspects in great detail. He also performed some Matlab simulations to determine average case performance. However, for a more in-depth evaluation of the algorithm under different network types in conjunction with routing protocols we need a standard simulation framework. Finally we have decided to use the industry standard network simulator NS2. This not only allows use to evaluate the topology management algorithm itself, but does enable us to evaluate different protocols and routing algorithms on top of NS2. We also identified possible problems in the practical implementation of the algorithm and instrumented the complete algorithm with debugging and statistical information. We put special focus on the graphical visualization of the algorithm, because this allows us to perform further research more easily than by looking at the internal data structures of the algorithm. Last but not least, we implemented an automated simulation framework to evaluate the algorithm for different network sizes. Such a semi automatic framework is important because any meaningful statistical results need at least a minimum size of different simulations.

1.4 Contribution and Key results

In this work we proved that the TMA is at least theoretically implementable in a real world environment although the network performance is not what one would expect from a practical algorithm. The performance problems arise mainly from the propose modules which perform a search-space exploration by means of exchanging messages and the periodic group checking with a message complexity of $O(|groups_{Max}|k^2)$. The maximum number of groups is given by $\lceil \frac{n-1}{k-1} \rceil$ [Tha05, p.14]. Being periodic, these components put a permanent pressure on the network capacity. These results are shown in Chapter 4. Nevertheless the resulting topology shows excellent characteristics with respect to power usage, fault-tolerance and its overall quality.

During our work we also gathered a lot of statistical information which allowed us to determine the components which warrant further investigation. This includes reducing the complexity of propose modules, and the average message complexity of the periodic group checking. An outlook for further research is presented in Section 1.6

1.5 Related Literature

A very good introduction to wireless networks is the classic paper by R. Rajaraman [Raj02]. It covers all basic concepts from modeling wireless networks, routing and topology management and contains a lot of useful references. More focused on topology management is the paper [San05] which contains a survey of state-of-the-art solutions for

topology management. The final reference for this work is the work of Thallner [Tha05] and Thallner, Moser [TM05].

The research interest in the routing protocol area is quite big and a lot of papers have been published. As an introduction the writer can recommend the paper [RT99] which covers a lot of basic adhoc routing protocols in some detail and serves as a good starting point. A very comprehensive survey is given in [Lan03].

The practical part of our work is focused on the network simulator NS2. A good starting point for working with the simulator is the NS2 manual [FV06] and the NS2 online resources [ud06]. If a more practical approach is preferred, a good paper is [RR04] which shows the implementation of a new routing protocol.

1.6 Further Research

During our studies we found that there are still a lot of open areas. Some of these areas are discussed in Chapter 5 where we adapted the network model to a more realistic one using bounded delays and lossy links. Using our framework we have developed the required infrastructure to conduct research in the following areas:

- The propose modules are the weakest part of our topology algorithm, because they account for most of the message complexity. For this purpose, additional propose modules should be implemented in the network simulator and their performance should be evaluated. This information can then be used to find the optimal mix between the quality of the topology and the message/time complexity for generating proposals.
- More research should be directed into the problem of a fully connected transmission graph to generate suitable network topologies. We have tried executing the algorithm in some not fully connected networks and the results where not satisfying because the final topology graph was partioned, i.e. it contained at least two components.
- All information from the topology management algorithm is available at the node. Some of this information could be used for routing protocols to improve their efficiency when they are used on top of the TMA. This somewhat reduces the strict separation of layers but could improve performance a lot. For example the routing protocol should never use its own “hello” protocol to discover its set of neighbors, because this information is already available.
- Some research should be performed to evaluate the dynamic properties of the TMA with moving nodes.

As fas as practical work is concerned, the following areas should be covered:

- The current wireless code of the network simulator NS2 is not very flexible. A recent patch has been proposed in [PH06] and the existing code should be ported to this new framework.

- Improving the performance of the current implementation on NS2. Profiling of the current implementation revealed that the weakest points are the slow relational operators for comparing group identifiers and groups.
- Implementing a real consensus algorithm for the NBAC - *Non-Blocking Atomic Commitment* problem together with a failure detector to allow for node crashes during group construction. The author is quite sure that these would further decrease performance of the algorithm, but having such a component within the simulator might be interesting and useful for other problems.

2 Topology Construction

2.1 Basics

An ad hoc wireless network consists of a set of wireless nodes, which communicate using a wireless network. Such a network can be modeled as a set of points in the Euclidean space where the communication between nodes is restricted by radio propagation and interference. Even if very simple models are used, describing a network becomes quite complex because the equations are non-linear, and if interference is taken into account, they contain a lot of variables. A good introduction is given in [Raj02, p.61]; we will repeat some of the basics here.

Whenever a node transmits a message using the transmission power P_t, the transmission is subject to path loss. A node listening on the wireless medium will receive the message with the receive power level P_r, which is a function of the distance d and a modeling parameter α, which abstracts physical parameters like antenna gain[1] and carrier frequency.

$$P_r = O\left(\frac{P_t}{d^\alpha}\right) \tag{2.1}$$

Note that equation 2.1 is very simplistic because it is a free-space model, i.e., there are no obstacles present. For a node to successfully receive a message, a given threshold β has to be reached. If no other nodes are currently transmitting, this is simply a function of noise N and the receive power level. A node y successfully receives a message from a node x_i transmitting with the power P_i if the threshold β is greater than $0.1-10$ [Raj02, p61]. This is shown in equation 2.2.

$$\frac{\frac{P_i}{d(x_i,y)^\alpha}}{N} \geq \beta \tag{2.2}$$

Typical values for alpha are $2-4$ [Raj02, p61]. Intuitively this is clear because if a message is transmitted with a given power level and the wave propagation is uniform in every direction the perimeter increases quadratically with the distance. If the power is distributed uniform the received power level obviously decreases with at least a quadratic exponent. Now assume that more than one node is transmitting. Let $T = \{x_1, \ldots, x_j\}$ be the set of transmitting nodes. Then a node y receives a message from node x_i if

$$\frac{\frac{P_i}{d(x_i,y)^\alpha}}{N + \sum_{k=1,k\neq i}^{j} \frac{P_k}{d(x_k,y)^\alpha}} \geq \beta \tag{2.3}$$

holds [Raj02, p61]. So other transmitting nodes simply increase the noise level for the message transmitted by node x_i. Although NS2 contains a realistic wireless model, this model is not useful for theoretical studies because it is too complicated. Therefore algorithm designers typically choose high level models like we did and prefer a more graph theoretic approach[2]. Examples for such models are shown in [Raj02, p62], [THB+02] and [CBD02].

[1] Antenna gain is the relative increase in [dB] to a standard basic antenna.

[2] For example, Wattenhofer in [WZ04] also modeled his network as a graph $G = (V, E)$.

An ad hoc network can be modeled as a graph $G = (V, E)$, where V represents the set of all nodes, and E is the set of all possible transmission links. Formally, such a graph is referred to as a *transmission graph* [Raj02, p62].

Definition 1. *Let V be the set of all nodes and E be the set of all direct communication links. For two nodes $x, y \in V$ we have $(x, y) \in E$ iff x can directly communicate with y. We call such a graph $G = (V, E)$ the* transmission graph *for a wireless network.*

Such a transmission graph for our example network is shown in Figure 2.1 which has been generated from within our NS2 simulation framework. It consists of $n' = 10$ "normal" wireless nodes and $n'' = 4$ gateway nodes. Normal wireless nodes are drawn using round shapes. Gateway nodes are drawn using rectangular shapes. Gateway nodes are always fully connected with each other over some dedicated backbone link. These nodes are required to guarantee that we can find a solution which is both k-connected and k-regular (See Definition 13). We can also see from this graph that all edges are associated with a weight, which is used as a metric for the cost of the link[3].

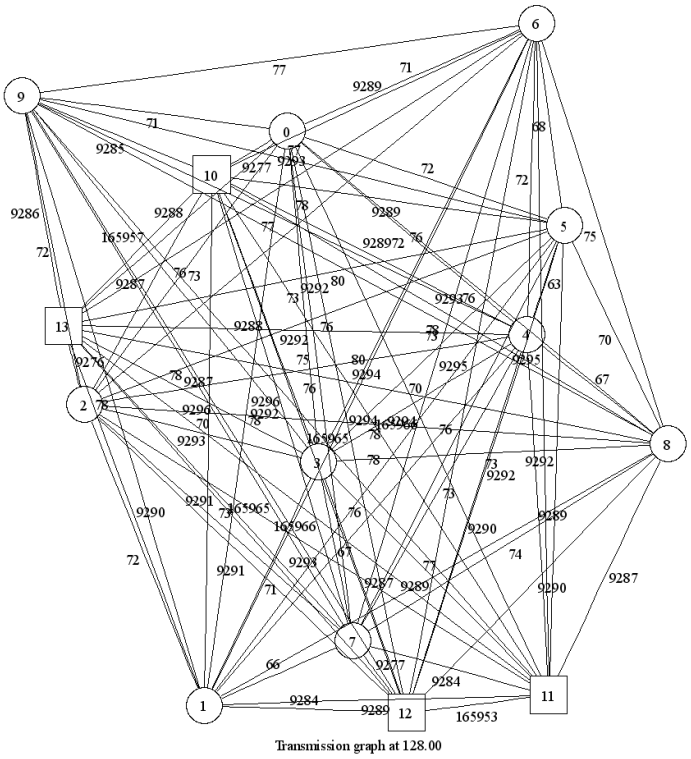


Figure 2.1: Transmission graph for a network with $n' = 10$ and $n'' = 4$

On the transmission graph we execute the Thallner algorithm as described in [Tha05].

[3] Or link costs are based on the path loss which is a function of the distance, the environment and the type of networking equipment (frequency, modulation, ...).

This algorithm constructs a low weight *overlay graph* or *topology graph* which is k-regular and k-connected.

Definition 2. *Let $d(v), v \in V$ be the node degree for an undirected graph. A graph $G = (V, E)$ is* k-regular *if all nodes have degree k, i.e. $\forall v \in V \Rightarrow d(v) = k$.*

Definition 3. *Let $G = (V, E)$ be the transmission graph. We say that two nodes x, y are directly connected, briefly $x \sim y$, if there exists an edge $(x, y) \in E$. Two nodes x, y are connected, briefly $x \sim^* y$, if there exists a sequence of nodes $x_0, x_1, \ldots, x_{n-1}, x_n$ with $x_0 = x$ and $x_n = y$ and $(x_{i-1}, x_i) \in E \quad 1 \leq i \leq n$. Note that $\sim^*$ is simply the transitive closure of $\sim$. If there is no direct respectively indirect communication we write $\nsim$ respectively $\nsim^*$.*

Definition 4. *A graph is k-node connected if there does not exist a set of $k-1$ vertices whose removal (and incident edges) disconnects the graph. Or more formally. Let $V' \subseteq V$ with $'V'| < k$ be the set of vertices which should be removed. Then for the graph $G' = (V', E') == (V \setminus V', E \setminus (V' \times V'))$ we have $\nexists x, y \in V' \left(x \nsim^* y\right)$ in G'.*

Definition 5. *Let $G = (V, E)$ be the transmission graph of an ad hoc wireless network. A* topology graph *$T = (V', E')$ is a graph where $V' = V$ and $E' \subseteq E$', such that if two nodes are connected in G, then they are also connected in T.*

An example of a topology graph is shown in Figure 2.2, which has been generated by executing the algorithm within the NS2 network simulator. The graph was built with $k = 3$ and the result shows the network topology after the algorithm has finished.

2.1.1 Network Model

During the first part of our work we will assume the following network model which will be relaxed in the second part of our work in Chapter 5. This network model is not very practical except for analytical studies, but some of the algorithms are not possible without these assumptions. For example is has been shown in [Gra78] that distributed systems with unreliable communication do not admit solutions to the Non-Blocking Atomic Commitment (NBAC) problem. Unfortunately the TMA requires multiple NBAC instances for group checking and group proposals. Nevertheless, some of these requirements can be simulated, for example by using retransmission algorithms. If some of these assumptions are violated and this is detected by our simulation framework, the simulation is aborted.

Asynchronous: We assume an asynchronous system, where no fixed upper bounds for message delivery are known, and no assumptions are made on the order or the time between computation events at any node.

Reliable links: Links are assumed to be reliable. A message that has been sent is eventually delivered.

Mobility: A node can change its position in the network at any time and to any location as long as the transmission graph remains connected.

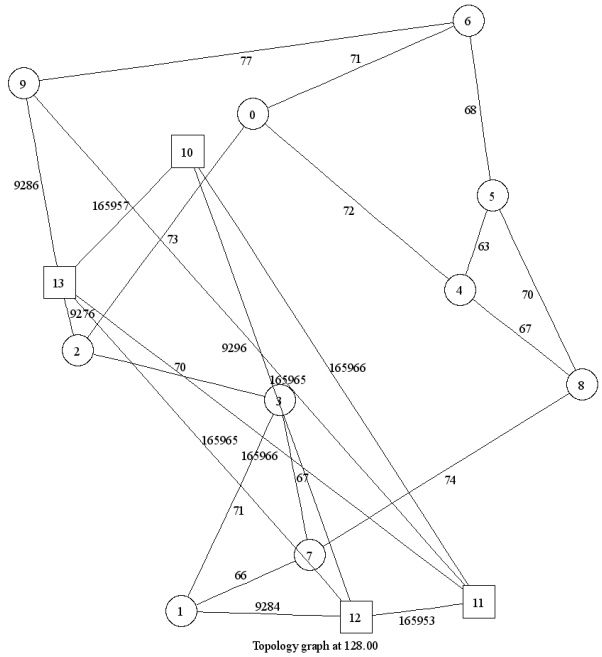


Figure 2.2: Topology graph for $k = 3$ and $n' = 10$ and $n'' = 4'$

Transmission graph: The network is modeled as a transmission graph $G = (V, E)$, where V is the set of nodes with $|V| = n$ and $n = n' + n''$, where n' are the normal nodes and n'' are the gateway nodes. E is the set of weighted edges $(x, y, w) \in E$ with $x, y \in V$ and $w \in \mathbb{R}^+$.

Fully connected: For every node $x, y \in V$ there exists an edge $(x, y, w) \in E$.

2.1.2 Topology Construction

The aim of the algorithm is to build a topology which is $k-$regular and $k-$connected. The basic idea behind the construction algorithm is to build groups of nodes that can subsequently be treated as a node. Such a group always consists of k members and always has k external connections left. For example looking at Figure 2.2 we can see that there exists a group with members $\{\{4\}, \{5\}, \{8\}\}$. This group is connected by the set of edges $\{(4, 5, 63), (4, 8, 67), (5, 8, 70)\}$. What is worth mentioning is that this group is a good choice with respect to the weight, because all other group formations for nodes $4, 5$ and 8 would be worse. The set of terminal nodes, which are by definition the nodes with one connection left, are $4, 5$ and 8. The set of terminal nodes are used to build the group id which in this case is $\{4, 5, 8\}$. This group can then subsequently be treated as a single node with k external connections. In our example these external connections are used to connect to other groups, as can be seen in Figure 2.2 by the link between 4

and node 0, the link between 5 and 6 and the link between 7 and 8. The resulting group $\{0, 6, 8\}$ therefore consists of the members $\{\{0\}, \{4, 5, 8\}, \{6\}\}$.

Because the complete group structure is not obvious from the graph, our simulation framework allows us to export the group membership information for every node. For node 6 this is shown in Figure 2.3. It is also possible to export a "global" view of the complete topology. This is shown in Figure 2.4. Note that in reality this is not possible because nodes do not possess this information, but within our simulation framework we can query all nodes for their local group information and then combine the topology trees from every node into a global tree[4].

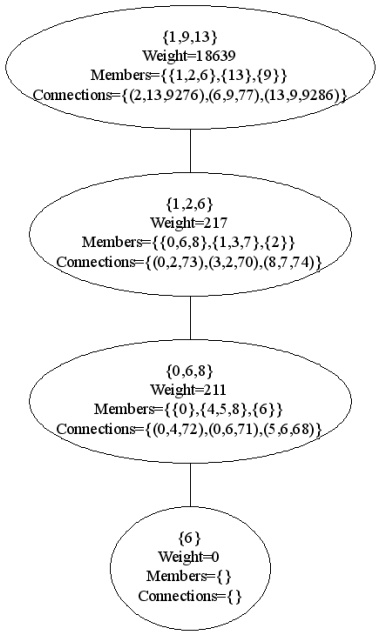


Figure 2.3: Local topology tree for node 6 for the topology shown in Figure 2.2

What we can see from Figure 2.3 is that the weight of the group $\{0, 6, 8\}$ is 211. A detailed explanation of the weight calculation will be given in Definition 12, but in this case it is simply the sum of the group internal connections. We also see that the group $\{0, 6, 8\}$ is a member in group $\{1, 2, 6\}$, and that in group $\{1, 2, 6\}$ node 6 is a terminal node. The reason for this is that the nodes 0 and 8 were used to build the group internal connections. This can be seen by looking at the group data structure for gid $\{1, 2, 6\}$ in Figure 2.3, where we see the internal connections $(0, 2, 73)$, $(3, 2, 70)$ and $(8, 7, 74)$. Hence only node 6 has a connection left and becomes one of the three terminal nodes. The weight of the group in this case is 217.
Another interesting thing we can see, is that the gateway nodes $10, 11, 12$ and 13 are treated differently and that their group weight is extremely high. For example the group $\{1, 9, 13\}$ has a group weight of 18639, which can never be exceeded by any group

[4]Note that if the network is currently under construction this is not guaranteed to be consistent.

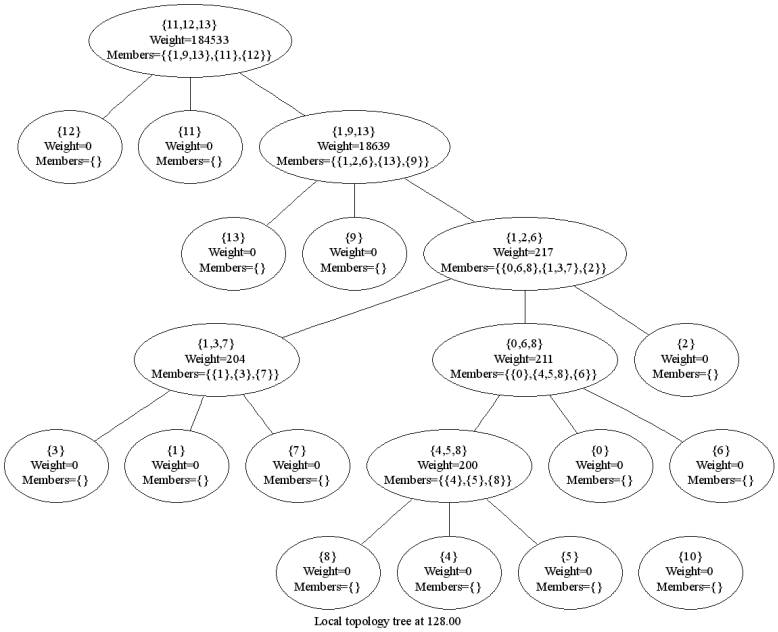


Figure 2.4: Topology tree for the topology shown in Figure 2.2

with non gateway nodes. This high weight results from the group internal connections $(2, 13, 9276)$, $(13, 9, 9286)$ and $(6, 9, 77)$. Group $\{11, 12, 13\}$ is even more expensive because it contains the connection $(11, 12, 165953)$ which happens to be a connection between gateway nodes. This group is built from the group internal connections $(1, 12, 9284)$,$(9, 11, 9296)$ and $(11, 12, 165953)$. A more complete explanation of the different connection weight classes is given in Definition 10.

We will now continue with a more formal description of our algorithm based on [Tha05]. For our purposes we have adapted the formalism used by Thallner to a simpler one.

Definition 6 (Nodes)**.** *Let $G = (V, E)$ be the transmission graph. We distinguish between the "normal" nodes V' with $|V'| = n'$ and the gateway nodes V'' with $|V''| = n''$ with $V = V' \cup V''$. Furthermore let* id *be an injective mapping* $\mathrm{id} : V \mapsto \mathbb{N}$*. This mapping associates every node with an unique integer id. If it is clear from the context we will sometimes directly use the integer id to identify the node.*

In our implementation the only practical difference is that connection between gateway- and gateway nodes and gateway-y and "normal" nodes are treated differently in the sense that connections to gateway nodes are very expensive and therefore they are not chosen by the algorithm. In any cases gateway nodes are assumed to be fully connected and these connections are static.

The example Figure 2.1 shows multiple nodes where every node has a unique id. For

example nodes 1 and 2 are "normal" nodes and the nodes $10, 11, 12$ and 13 are gateway nodes. It is important that $n'' \geq 2k - 2$ [Tha05, p15] to ensure that the final graph is $k - connected$.

Definition 7 (Groups)**.** *Let $\mathbb{G}$ be the set of group identifiers consisting of the $k-$member group identifiers $\mathbb{G}' = \mathrm{id}(V)^k$ and the single-node group identifiers $\mathbb{G}'' = \mathrm{id}(V)$. Every group consists of a group identifier $gid \in \mathbb{G}$, 0 or k members, up to $k(k-1)/2$ group internal connections and a group specific weight.*
Every non-single node group has exactly k, members where a member is either a single node group or again is a group. By definition single node groups have no members. We identify the set of members of a group gid with members(gid)*. Furthermore every node or group can only be a member of a single group:*

$$x \in \mathrm{members}(g_i) \Rightarrow (\forall g_j \in \mathbb{G}, g_i \neq g_j \Rightarrow x \notin members(g_j))$$

Valid group identifiers in Figure 2.4 are for example $\{4, 5, 8\} \in \mathbb{G}'$ or $\{8\} \in \mathbb{G}''$. The curly braces around the group identifiers should not be confused with the ones used for sets. They are only used for presentation purposes and if used otherwise it is specially noted. An example for the members mapping is $\mathrm{members}(\{0, 6, 8\}) = \{\{0\}, \{4, 5, 7\}, \{6\}\}$ or $\mathrm{members}(\{6\}) = \{\}$.

Definition 8. *The* level l members *are defined recursively as:*

$$\begin{aligned} \mathrm{members}^0(gid) &= \mathrm{members}(gid) \\ \mathrm{members}^j(gid) &= \bigcup_{r \in \mathrm{members}^{j-1}(gid) \wedge r \in \mathbb{G}'} \mathrm{members}(r) \end{aligned}$$

Or in words. the level l members for the group gid are the members of the groups which are l levels deeper in the topology tree.

For example let us use the group membership information from Figure 2.4.

$$\begin{aligned} \mathrm{members}^0(\{1,2,6\}) &= \{\{0,6,8\}, \{1,3,7\}, \{2\}\} \\ \mathrm{members}^1(\{1,2,6\}) &= \bigcup_{r \in \mathrm{members}^0(\{1,2,6\}) \wedge r \in \mathbb{G}'} \mathrm{members}(r) \\ &= \mathrm{members}(\{1,3,7\}) \cup \mathrm{members}(\{0,6,8\}) \\ &= \{\{1\},\{3\},\{7\}\} \cup \{\{0\},\{4,5,8\},\{6\}\} \\ &= \{\{1\},\{3\},\{7\},\{0\},\{4,5,8\},\{6\}\} \\ \mathrm{members}^2\{1,2,6\} &= \bigcup_{r \in \mathrm{members}^1(\{1,2,6\}) \wedge r \in \mathbb{G}'} \mathrm{members}(r) \\ &= \mathrm{members}(\{4,5,8\}) \\ &= \{\{4\},\{5\},\{8\}\} \\ \mathrm{members}^3\{1,2,6\} &= \emptyset \end{aligned}$$

Definition 9. *The set of nodes for a group gid is defined as:*

$$\text{nodes}(gid) \quad = \quad \bigcup_{l=0}^{\infty} \text{members}^l(gid) \cap V$$

By convention we define $\text{nodes}(v) = \{v\}$ *for* $v \in V$.

Using the previous example this gives us:

$$\begin{aligned} \text{nodes}(gid) &= \{\{0,6,8\},\{1,3,7\},\{2\},\{1\},\{3\},\{7\},\{0\},\{4,5,8\},\{6\},\{4\},\{5\},\{8\}\} \cap V \\ &= \{\{2\},\{1\},\{3\},\{7\},\{0\},\{6\},\{4\},\{5\},\{8\}\} \end{aligned}$$

Definition 10 (Connection)**.** *A connection* $(x, y, \omega) \in E$ *is a triple where* $x, y \in V$ *and* $\omega \in \mathbb{R}^+$. *It is called a* group *gid* internal connection *if* $x \in \text{nodes}(g_1)$ *and* $y \in \text{nodes}(g_2)$ *with* $g_1, g_2 \in \text{members}(gid)$ *and* $g_1 \neq g_2$. *We denote these connections by* $\text{internal}(gid)$. *Furthermore we impose some restrictions on the connection weight for a connection* $(x, y, \omega) \in E$. *Let* K *be an arbitrary but existing constant in* $\mathbb{R}^+$.

$$\begin{aligned} x, y \in V' &\Rightarrow \omega \leq K \text{ (connection between normal nodes)} \\ \begin{matrix}(x \in V' \wedge y \in V'') \\ \vee \\ (x \in V'' \wedge y \in V')\end{matrix} &\Rightarrow 2k^2K \leq \omega \leq 4k^4K \text{ (connection between gateway and normal node)} \\ x, y \in V'' &\Rightarrow 4k^4K \leq \omega \text{ (connection between gateways)} \end{aligned}$$

The restrictions on the weight are required to ensure that gateway nodes are not preferred during group constructions. Furthermore, the artificially high weight for gateway internal connections[5] prevents the algorithm from building gateway groups. The definition by itself is consistent with that from Thallner in [Tha05, p9] with the exception of our connections between gateways[6].

For example in Figure 2.2 the connection $(0, 2, 73)$ is a group internal connection for $\{1, 2, 6\}$ because $0 \in \text{nodes}(\{0, 6, 8\})$ and $2 \in \text{nodes}(\{2\})$ and $\{0, 6, 8\}$ and $\{2\} \in \text{members}(\{1, 2, 6\})$. Other group internal connections are for example $(3, 2, 70)$ and $(8, 7, 74)$ in $\{1, 2, 6\}$. All group internal connections are available from our simulation framework in NS2. The example above was extracted from Figure 2.3.

Similar to the connectivity between nodes defined in Definition 3, we can define a connectivity between groups.

Definition 11 (Connectivity)**.** *Let* $g_1, g_2 \in \mathbb{G}$. *We call the groups* directly connected *if there exists a connection* (x, y, w) *with* $x \in \text{nodes}(g_1)$ *and* $y \in \text{nodes}(g_2)$. *If two groups* g_1, g_2 *are directly connected we will write* $g_1 \sim g_2$ *as an abbreviation. For a connection on the transitive closure of the relation* $\sim$ *we will write* $g_1 \sim^* g_2$.

Looking at our example in Figure 2.3 and 2.2 we can see that the groups $\{4, 5, 8\}$ and $\{0, 6, 8\}$ are connected by the edge $(5, 6, 69)$, i.e., $\{4, 5, 8\} \sim \{0, 6, 8\}$. The members of a group are always fully connected. Furthermore we can see that $\{1, 3, 7\} \sim^* \{6\}$ because $\{1, 3, 7\} \sim \{0, 6, 8\}$ and $\{0, 6, 8\} \sim \{6\}$, but $\{1, 3, 7\} \nsim \{6\}$

[5] This results in our implementation from the fact that all nodes are treated equal.

[6] Thallner did not require this because he used a different implementation for gateway nodes.

Definition 12 (Group weight)**.** *The weight of a group* $\omega(gid)$ *is a triple* $(A, \text{members}(gid), \text{internal}(gid))$ *where*

$$A = \max\left(\sum_{(x,y,w)\in\text{internal}(gid)} w, \max_{\substack{\omega(g_i)=(A_i,\dots),\\ g_i\in\text{members}(gid)}} (A_i) + \epsilon\right)$$

with some $\epsilon > 0$*. By definition the weight of a single node group is alway zero.*

So the weight of a group is either the sum of its internal connections or the maximum weight of any of its members plus an arbitrary small constant. Note that the weight of a parent group is always greater than the weight of any of its members. To see an example for these two cases we have provided an additional example in Figure 2.5. We can see that the group $\{3, 5, 9\}$ has a group weight of 204.1. The reason for this is that the sum of the group internal connections is 203, which can be seen in Figure 2.6(a), but the weight of group $\{2, 3, 7\}$ is 204. In our case $\epsilon = 0.1$ and therefore the group weight becomes 204.1. An example where the sum of internal connections is used is the group $\{2, 3, 7\}$, where the weight is simply $68 + 71 + 65 = 204$. The associated topology graph is shown in Figure 2.6(b) to finish this example.

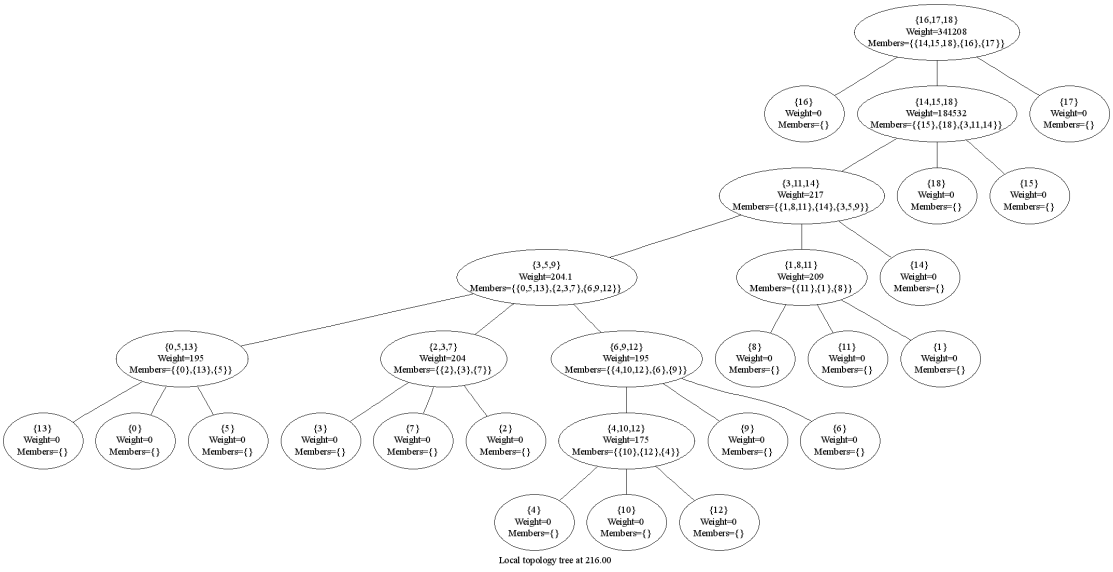
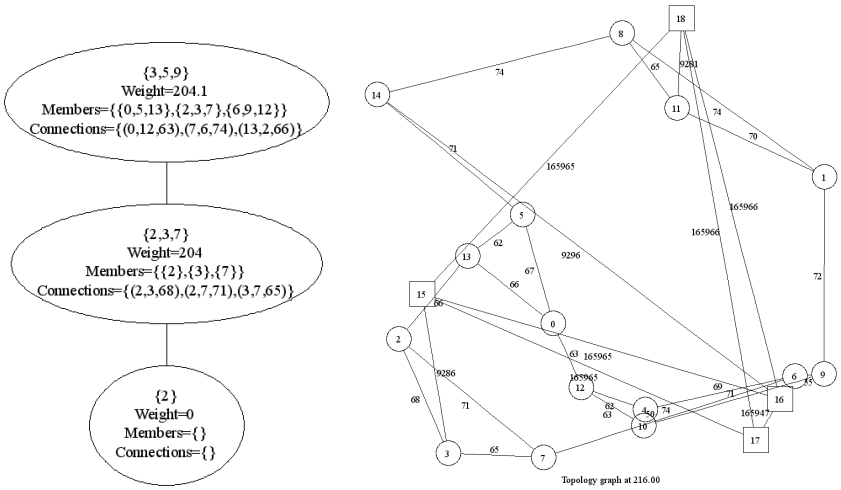


Figure 2.5: Topology tree node $n' = 15$ and $n'' = 4$.

Definition 13. *An topology graph is called* admissible *if the topology consists of a single root group where all terminal nodes are gateway nodes.*

Looking at our example in Figure 2.4 we see that there exists a single top level group $\{11, 12, 133\}$ and all normal nodes are 3−connected. Please note that in contrast to the original Thallner version, we have chosen to implement the gateway nodes slightly



(a) Local topology tree for node 2

(b) Topology graph

Figure 2.6: Topology graph and local topology tree for node 2 for the topology shown in 2.5

differently in NS2: Gateway nodes are like normal nodes, but always have connections between them which are not affected by the execution of the algorithm. Anyway this is somewhat similar to Thallner who used a dedicated backbone [Tha05, p9]. Note that it is also possible and allowed using the definition above that not all gateway nodes are used in the final topology.

Definition 14 (Topology tree)**.** *The* topology tree *is the graph constructed by recursively adding all group members to the graph, starting with the root group. By definition the* root group *is the group which is not a member of any other group* $g \in \mathbb{G}$ *and it has* k *members*[7]*. For two groups* g_1, g_2 *there exists a connection in the topology tree when* $g_1 \in \text{members}(g_2)$*. Because every group has* k *members this gives us a* $k-$*ary tree.*

The topology tree for our example in Figure 2.2 is shown in Figure 2.7. We start with the root group $\{11, 12, 13\}$, which has the members $\{11\}$, $\{12\}$ and $\{1, 9, 13\}$. Therefore these members are also added to the graph and a connection is drawn to the parent group. This process is continued until all groups have been processed.

2.1.3 Properties

We will just repeat the most important properties of the Thallner algorithm and will not give any of the proofs. The interested reader is referred to [Tha05] and [TM05] where they are covered in depth.

[7] The last addition is required because gateway nodes are treated in the same way as normal nodes and they might be "left over" at the end of execution.

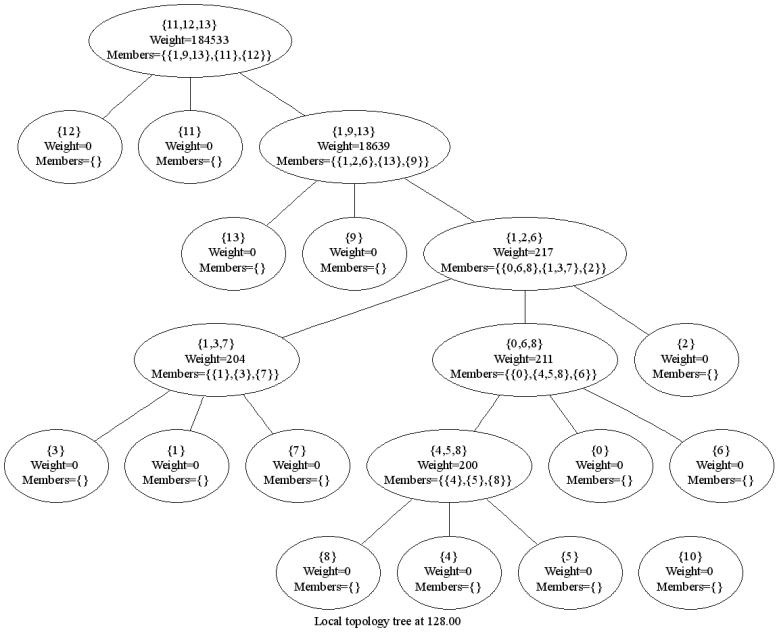


Figure 2.7: Topology tree for the topology shown in Figure 2.2

Completeness: For every transmission graph $G = (V, E)$ there exists an admissible overlay graph if $n' \geq 1$ and $n'' \geq 2k - 2$ [Tha05, p13].
This implies that one should take care when starting a simulation that there are always enough gateway nodes. For example for $k = 2$ there need to be two gateway nodes. For $k = 3$ four gateway nodes are sufficient.

k-regularity: In every admissible overlay graph G, the node degree is bounded by k and all normal nodes have degree k [Tha05, p14].
Note that connections between gateways do not count in this definition. This property also implies that although the algorithm has converged, there might still be gateway nodes left which have a connection "left".

Number of groups: Every admissible overlay graph with $n' \geq 1$ and $n'' \geq 2k - 2$ has *exactly* $\left\lfloor \frac{n-1}{k-1} \right\rfloor$ groups [Tha05, p14].

Relation between Topology Tree and Topology/Overlay Graph: Every admissible overlay graph corresponds to a unique topology tree for $k > 2$. This means that if the groups are fixed, then the overlay graph is fixed, and vice versa [Tha05, p12].
So given the topology tree one can draw the topology graph and vice versa.

Uniqueness of minimal Overlay Graph: There exists exactly one minimal admissible overlay graph for every graph G with $n' \geq 1$ and $n'' \geq 2k - 2$ [Tha05, p13].

We added some custom (and simple) properties which are used later in our work.

Theorem 1 (maximum height)**.** *The maximum height of a topology tree is* $\lceil \frac{n-1}{k} \rceil$.

This metric is important because it can be used to estimate an upper bound on the time required for checking the complete topology which is used in our convergence detection algorithms.

Proof. Let n be the number of groups and let k be the "aryness" of the tree. The Thallner algorithm only build groups of k members. Therefore any non leaf node must have exactly k-children. Let us start with a root node. We can now add up to k children to this node. Adding more nodes would require us to again add k children to any of the previous children. Because we want our result to hold for the deepest tree we will always choose the child which is deepest in the tree constructed so far.

The proof is by induction on the number of nodes n where our induction step is k and not 1. The induction basis is given for $n = 1$ and $n = k + 1$. For $n = 1$ it is obvious by drawing the tree which shows a depth of 0. For $n = k + 1$ we have a root node with k children which gives a depth of 1. Again this fits our claim. Now assume our result holds for $n = 1 + ck$ where $c \in \mathbb{N}$. By the hypothesis the deepest node has depth c. This node has no children and if we add k children we increase the depth by one. Now $n' = n + k = 1 + ck + k$ is $n' = 1 + (c + 1)k$ and therefore $\left\lceil \frac{(1+(c+1)k)-1}{k} \right\rceil = c + 1$ giving us our desired result. □

2.2 Examples

In this chapter we will show some concrete examples of how the TMA is supposed to work, with a focus on the distributed implementation shown in Chapter 2.3. We will skip most of the details, which are relayed to the next chapters, but the basic idea should be obvious after reading this chapter. We start with fully transmission graph as shown in Figure 2.8.

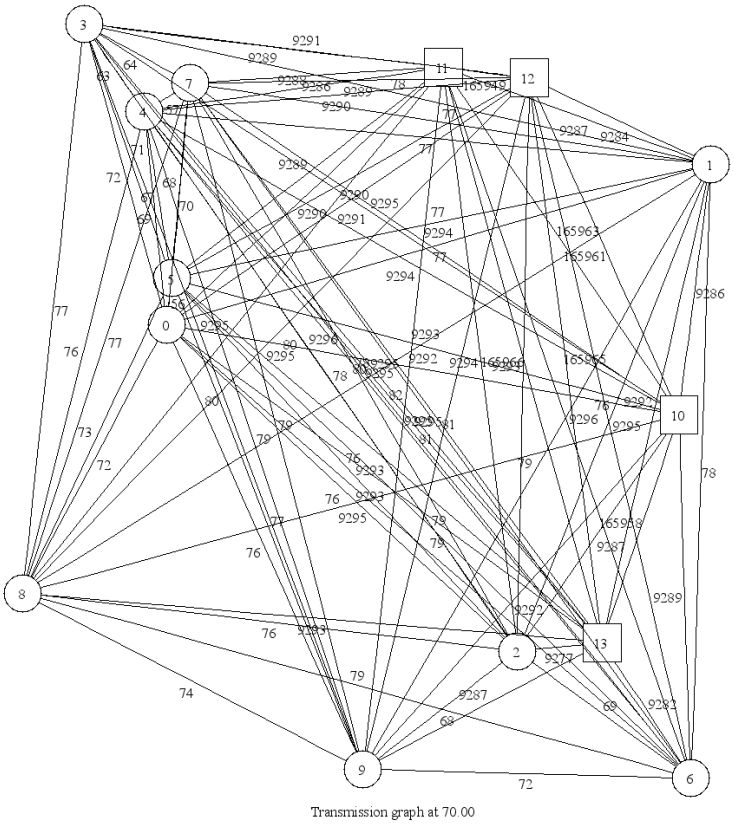


Figure 2.8: Example transmission graph for a network with $'n = 10$ and $n'' = 4$

The algorithm consists of two independent modules. The propose modules, which we will treat in Section 2.2.1, and the group checking, which we cover in Section 2.2.3.

2.2.1 Generation of Proposals

The primary goal of the TMA is to build new groups and to check that already built groups are still alive and/or are a good choice with respect to some metric defined upon weights. New groups are suggested by so called *propose modules*. Such a module generates a group proposal, which includes at least the members, the group internal connections, the set of k terminal nodes and a weight. For now we can simply assume that

there exists such a *propose module* which must satisfy at least the following properties:

- A propose module generates proposals for groups of k members with k terminal nodes, group internal connections and the corresponding group weight [Tha05, p17].
- Every propose module generates infinitely many proposals.
- Every node has access to such a propose module.

Definition 15 (Group Proposal)**.** *A group proposal is a quadruple*

$$Prop = (Members, Terminal\ nodes, Internal\ connections, Weight)$$

where $Members$ *is a set of* k *group IDs, i.e.* $\{gid_1, \ldots, gid_k\}$ *with* $gid_1, \ldots, gid_3 \in \mathbb{G}$*. The terminal nodes are the nodes with one connection left in the group and at the same time are used as the group id. Therefore* $\mathrm{gid}(Prop) = \mathrm{terminals}(Prop) = \{\mathrm{id}(v_1), \ldots, \mathrm{id}(v_3)\}$ *with* $v_1, \ldots, v_3 \in V$*. There are exactly* $k(k-1)/2$ *group internal connections with* $(v_{i_1}, v_{i_2}, \omega) \in E$ *with* $1 \leq i \leq k$ *and* $v_{i_1}, v_{i_2} \in \mathrm{terminals}(gid_1) \cup \mathrm{terminals}(gid_2) \cup \mathrm{terminals}(gid_3)$ *The group internal weight is defined according to Definition 12.*

We have already defined what a group is in the Definitions 7, 10 and 12. For simplicity we will adopt the same notation as for the group proposals given in Definition 15. For algorithm simplicity we allow single node groups to exist at the bottom level. Therefore for every node we define:

Definition 16 (Group)**.** *For every node* $v \in V$ *there exists a group* $(\{\mathrm{id}(v)\}, \{\mathrm{id}(v)\}, \emptyset, 0)$

Definition 17 (Parent and Ancestor Relation for Groups)**.** *We introduce another relation* parent : $\mathbb{G} \mapsto \mathbb{G}$ *where* $\mathrm{parent}(gid_i) = gid_j$ *iff the parent group of* gid_i *is* gid_j*. Furthermore we define* ancestor *as the transitive closure of the relation* parent *and write* $\mathrm{ancestor}(gid_\ell)) = gid_j$ *if there exists a sequence of groups* $gid_\ell), gid_{\ell)+1}, gid_{\ell)+2}, \ldots, gid_j$ *with* $\mathrm{parent}(gid_\ell) = gid_{\ell)+1}$ *for* $1 \leq \ell) < j$*.*

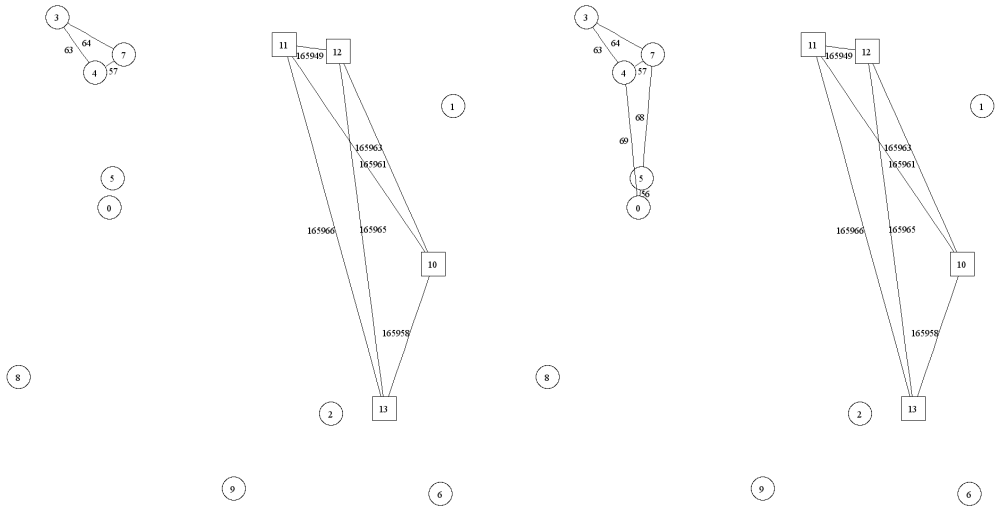
A valid proposal for our transmission graph in Figure 2.8 would be

$$Prop_1 = (\{\{3\}, \{4\}, \{7\}\}, \{3, 4, 7\}, \{(3, 7, 64), (3, 4, 63), (4, 7, 57)\}, 184)$$

where in this case the set of members are the same as the terminal nodes. We can also see that there are exactly 3 group internal connections corresponding to $k = 3$. Furthermore the group weight is consistent with Definition 12 because the weight of single nodes is zero. Therefore the largest weight of all its members is zero and the sum of group internal connections makes up the weight which happens to be 184. The resulting topology graph is shown in Figure 2.9(a). After this group has been created the search space is "expanded" and a new possible proposal is for example

$$Prop_2 = (\{\{3,4,7\},\{0\},\{5\}\},\{0,3,5\},\{(0,5,56),(0,4,69),(5,7,68)\},193).$$

The "after"[8] implies that the group construction starts at the bottom where groups of nodes are formed. These groups of nodes are then treated as "super-"nodes which are again used to build groups. The weight of the group $\{3,4,7\}$ happens to be 193 because the sum of group internal connections is bigger than the weight of any of its members. The result of adding this group to the topology is depcted in Figure 2.9(b).



(a) Groups $\{3,4,7\}$ built.

(b) Groups $\{3,4,7\}$ and $\{0,3,5\}$ built.

Figure 2.9: Example for proposals and group formations for $n' = 10$ and $n'' = 4$.

Typically proposals are generated using "simple" search strategies [Tha05, p57]. Such search strategies are implemented by *Propose Modules* which are covered in depth in [Tha05, p57-82]. These modules can be classified with respect to their search space and to their level of distribution. For the *search space exploration* there are two classes. The first class are the *perfect propose modules* which explore the whole search space.

Definition 18 (Perfect propose module)**.** *A propose module is called* perfect *if it eventually generates a proposal in the final minimum admissible topology graph infinitely often [Tha05, p57].*

If every node uses perfect propose modules it is guaranteed that the minimal admissible topology graph is constructed [Tha05, p57]. The second class are the *non-perfect propose modules.*

Definition 19 (Non-perfect propose module)**.** *Non-perfect propose modules have a restricted search space [Tha05, p57].*

[8] This is a direct result of the local propose module which always starts with an already existing group. At the startup these are of course the single node groups.

These modules cannot guarantee that the minimal topology is found, but only that an admissible topology graph is found. An example for such a module is the *Local-Non-Perfect Propose Module* which always generates proposals containing itself, and only forwards it to its lowest-weight edge not visited before [Tha05, p67]. We can also classify propose modules with respect to their distribution.

Definition 20 (Global propose module)**.** *A propose module is a* global *propose module if it uses some kind of coordinator to serialize group construction [Tha05, p57].*

Such propose modules can guarantee that groups are never destroyed by later group constructions [Tha05, p58].

Definition 21 (Local propose module)**.** *A* local *propose module is fully distributed.*

A local propose module might build suboptimal groups because it does not have the complete information. Therefore some groups might be destroyed later because a group (in fact the decision is taken by the terminal nodes of these groups) decides to join a newly proposed group and leaves its current group.

Definition 22 (Locally-agreed propose module)**.** *A* locally agreed *propose module performs the construction as concurrent as possible but ensures that groups built are not destroyed by later constructions.*

It is important that convergence and correctness of the group construction algorithm does not depend on the propose modules. But of course propose modules are crucial for liveness and performance of the complete system [Tha05, p57].

2.2.2 Emission of proposals

We have already talked about what proposals are and how proposals are used to build groups. A remaining question is how nodes in a fully distributed system decide wether they want to accept a proposal and if yes, how to join such a newly proposed group. The general idea is as following:

- A node periodically triggers its propose module to generate new proposals and waits for *released* proposals. A released proposal is simply a valid proposal found by a propose module.
- If such a proposal is received by a message the node initiates a special protocol, which enforces consensus on whether to join the group or not. All terminal nodes of all members in this proposal take part in this protocol. The reason why all terminal nodes must take place is that all these nodes have to update their group information about the possible new parent group (if accepted). For example in the proposal

$$Prop_2 = (\{\{3,4,7\},\{0\},\{5\}\},\{0,3,5\},\{(0,5,56),(0,4,69),(5,7,68)\},193)$$

the nodes $0, 3, 4, 5$ and 7 will participate. Note that there are at most k^2 participants because there can only be k-members, and every member is either a group consisting of k terminal nodes or a single node.

- Using this list of participants the initiator initiates a *Non-blocking atomic commitment* (NBAC) protocol which guarantees that all nodes either agree on the new proposal, also called **COMMIT**, or disagree on the proposal, also called **ABORT**.
- If a node receives a proposal it decides locally whether to join or not to join the newly proposed group. A group is joined if it is better, that is, if its weight is smaller thans its current parent group weight, or if the node has no parent group at all. The result of the voting process is either **COMMIT** or **ABORT**. This vote is then broadcast to all other participants.
- If all nodes have received all votes or if a node has been suspected, a consensus protocol is used to ensure that all nodes agree on the decision. Node suspection is typically implemented by a failure detector. After this step has been taken a node locally decides and if the decision is **COMMIT**, it updates its local group membership information and marks the new group as temporary. This is required to prevent it from being destroyed by the parallel group checking algorithm because not all nodes might decide at the same time.
- After all nodes have decided every node finalizes the decision and the groups temporary status is removed. Now the group information is the same at every correct node and the group will be checked by the group checking algorithm to ensure its consistency.

The complete code for the group proposal algorithm is shown in Chapter 2.3 in Listing 2.6.

2.2.3 Periodic Group Checking

A wireless adhoc network is not a static structure and therefore the set of vertices and edges is potentially time variant. In addition groups might decide to leave their current group and decide to join a new group. This could possibly render an old group inconsistent. It is therefore necessary to periodically check the groups for consistency. Again some sort of atomic commitment protocol is used because all nodes should have the same view of the overlay graph and should reach a common agreement on whether the group should be destroyed or is still a good (and consistent) one.

The approach is a polling one, which has the drawback of generating a constant "background" traffic but it guarantees that network changes and nodes crashes can be detected easily. Looking at our example in Figure 2.9(b) node 3, knows about the groups $\{3\}$, $\{3, 4, 7\}$ and $\{0, 3, 5\}$ where they have the following structure:

$$\begin{aligned} \{3\} &= (\{3\}, \{3\}, \emptyset, 0) \\ \{3,4,7\} &= (\{\{3\}, \{4\}, \{7\}\}, \{3,4,7\}, \{(0,5,56), (0,4,69), (5,7,68)\}, 193) \\ \{0,3,5\} &= (\{\{3,4,7\}, \{0\}, \{5\}\}, \{0,3,5\}, \{(0,5,56), (0,4,69), (5,7,68)\}, 193) \end{aligned}$$

Basically every node $v \in V$ periodically checks all groups $gid \in \text{ancestor}(\text{id}(v))$. The general algorithm for group checking is as following:

- A node waits for a periodic trigger signal. Then it recursively walks up its local group hierarchy using the parent relation and checks every group[9]. Groups which are locked, i.e. marked as temporary be the group proposal algorithm, are not checked.
- For every group g it initiates an atomic commitment protocol with all terminal nodes of members(g). The payload of the message is the current group information.
- Every node participating in the atomic commitment protocol receives this group information and checks if it is consistent with its local information. It is consistent if it has the same set of members and connections. Depending on the result it returns either **COMMIT** or **ABORT**. The votes together with the node's local connection information (which might have changed since the construction), is broadcast to all other nodes.
- After all votes from not suspected nodes are available, an agreement protocol is executed. After this all correct nodes have the same vote.
- Now a decision procedure is invoked. If the common decision is **ABORT**, the group is destroyed. Destroyed means that this group and all its parent groups are left. If the decision is **COMMIT**, the weight information is recalculated from the payload in the votes. In this case the weight of the group, which might have changed, is checked for consistency according to Definition 12. Note that this last step might lead to an inconsistent view of the network, but during the next group checking all nodes would see the change because nodes which left would vote **ABORT**.

The complete code for the group checking is shown in Section 2.3.5 in Listing 2.3.5.

[9] The actual implementation is somewhat different in that it does not check all groups at once, because this introduces too much load on the network if all instances are done in parallel. It therefore checks the groups in a round robin fashion.

2.3 Distributed Construction Algorithm

In this section we will show how the Thallner algorithm can be implemented. This section is based on the work of Thallner found in [Tha05] with some additional explanations and modifications for the atomic commitment implementation.

2.3.1 Data structures

We have already introduced a mathematical description for groups in Definition 7 and for group proposals in Definition 15. These mathematical descriptions are not very suitable for describing the algorithm and therefore we would like to introduce some additional notations. The mapping to equivalent C++ classes is straightforward and has been performed in the file `tma-types.h`.

Definition 23 (Group Structure)**.** *A* group data structure *is a record like type which contains the fields shown in Listing 2.1.*

Listing 2.1: Group record

$$
\begin{aligned}
&Record\ Group\ is\ \{\\
&\quad Members\ :\ Set\ of\ k\ group\ or\ node\ IDs.\ Members \subseteq \mathbb{G}, |Members| = k\\
&\quad Weight\ :\ Weight\ of\ the\ group.\ Weight \in \mathbb{R}^+\\
&\quad Connections\ :\ Set\ of\ connections.\ Connection \subseteq E, |Connection| = k\ or\ 0\\
&\quad Terminals\ :\ The\ terminals\ nodes.\ Terminals \in \mathbb{G}\\
&\}
\end{aligned}
$$

For a group *Group* we will write $Group.Members$ if we want to refer to the $Members$ field of this record. It must be noted that storing the weight as an real value is only sufficient because the other required information enforced by Definition 12 is also available. This is necessary because if two groups have the same weight the members and connections are used to enforce a strict ordering on the groups.

Definition 24 (Group Internal Structure)**.** *A* group internal structure *is used to represent the topology tree at a node. It contains additional fields for locking the groups during construction and a member* $Group.ParentID$ *to refer to the parent group. It is shown in Listing 2.2.*

Listing 2.2: Group internal record

$$
\begin{aligned}
&Record\ Group\_Internal\ is\ \{\\
&\quad Members\ :\ Set\ of\ k\ group\ or\ node\ IDs.\ Members \subseteq \mathbb{G}, |Members| = k\\
&\quad Weight\ :\ Weight\ of\ the\ group.\ Weight \in \mathbb{R}^+\\
&\quad Connections\ :\ Set\ of\ connections.\ Connection \subseteq E, |Connection| = k\ or\ 0\\
&\quad Terminals\ :\ The\ terminals\ nodes.\ Terminals \in \mathbb{G}\\
&\quad ParentID\ :\ Group\ ID\ of\ the\ parent\ group.\ ParentID \in \mathbb{G}\\
&\quad LockedBY\ :\ Unique\ NBAC\ ID\ for\ temporary\ locking.\\
&\}
\end{aligned}
$$

Definition 25 (Connection Structure)**.** *A* connection *is a triple containing a source, a destination node and an arbitrary weight. It is mapped to the record data type shown in Listing 2.3.*

Listing 2.3: Connection record

```
Record Connection is {
  x : Source node. x ∈ V
  y : Destination node. y ∈ V
  ω : Weight. ω ∈ ℝ+.
}
```

2.3.2 Main loop

The main loop waits for new proposals from the propose module in line $15-18$ which are then released to the network by initiating the atomic commitment protocol. Messages from the atomic commitment protocol are processed in lines $11-13$. In addition every node has access to a periodic timer which triggers the generation of new proposals in lines $21-22$ and the checking of the groups in lines $24-30$.

Listing 2.4: Main loop

```
1  var Group[]
2
3  // {ID} is the node locals id
4
5  Group[{ID}].Members ← {}
6  Group[{ID}].Weight ← 0
7  Group[{ID}].Connections ← {}
8  Group[{ID}].ParentID ← ⊥
9  Group[{ID}].LockedBy ← ⊥
10
11 upon receival of NBAC message
12   if received atomic commitment message
13     execute atomic commitment phase
14
15 upon receival of new proposal P from propose module
16   initiate atomic commit PROPOSE_GROUP
17     participants ← terminal nodes of P.Members
18     data ← P
19
20 upon signal
21   if received signal to generate proposal
22     trigger propose module
23
24   if received signal to check groups
25     gid ← Group[ID].ParentID
26     while gid ≠ ⊥ ∧ Group[gid].LockedBy ≠ ⊥
27       initiate atomic commit CHECK_GROUP
28         participants ← terminal nodes of Group[gid].Members
```

```
29            data ← Group[gid]
30          gid ← Group[gid].ParentID
```

2.3.3 Utility functions

These functions are more or less self explanatory but some things should be mentioned for easier understanding. If one looks at the calculate_weight function it should be obvious that it returns a group weight according to Definition 12. Furthermore we require to treat the sets in lines $40 - 41$ as multisets.

Listing 2.5: Utility functions

```
31 function is_locally_consistent(group)
32   var gid ← group.Terminals
33   return\
34     (Group[gid] ≠ ⊥)∧ // Group exists locally\
35     (Group[gid].Members = group.Members)∧ // Group has the same members \
36     (Group[gid].Connections = group.Connection) // Group has the same connections
37
38 function calculate_weight(data, group)
39   // calculate weight for group from commit data.
40   var Group_Weights = {ω_Group : (ω_Group, ω_Conns) ∈ data}
41   var Conn_WeightSums = {ω_Conns : (ω_Group, ω_Conns) ∈ data}
42   return (max(∑ Conn_WeightSums, ∑ Group_Weights + ϵ), group.Members, group.Connections)
43
44 function join_group(gid, group)
45   var pgid ← group.Terminals
46   // group gid leaves current parent group and joins new group.
47   if Group[gid].ParentID ≠ ⊥
48     leave_group(gid)
49   // create the new parent group locally.
50   Group[pgid] ← group
51   // update the current group parent to point to the newly joined group.
52   Group[gid].ParentID ← pgid
53   // create the connection in the topology graph.
54   for c ∈ Group.Connections
55     if ID is in c
56       make_connection(c)
57
58 function leave_group(gid)
59   // group gid recursively leaves its current parent groups.
60   if Group[gid].ParentID ≠ ⊥
61     var pgid ← Group[gid].ParentID
62     // cancel all connections in the topology graph.
63     for c ∈ Group[pgid].Connections
64       if ID is in c
65         cancel_connection(c)
66     // recursively leave parent groups of the current parent.
67     leave_group(pgid)
```

```
68      // this group has no parent anymore.
69      Group[gid].ParentID ← ⊥
70
71  function want_to_join(gid, group)
72    // does group gid want to join group?
73    if group[gid] ≠ ⊥
74      var pgid ← Group[gid].ParentID
75      var new_pgid ← Group[group].Terminals
76      return
77        // We don't have a parent group or the new group is better
78        // than the current parent.
79        (pgid = ⊥ ∨ group.Weight < Group[pgid].Weight) ∧
80        // The new parent group must either not exist and if it
81        // exists it must satisfy our weight constraint.
82        (Group[new_pgid] = ⊥ ∨ Group[gid].Weight < Group[new_pgid].Weight) ∧
83        // The new group must have a higher weight than our group.
84        Group[gid].Weight < group.Weight
85    else
86      // the group gid is no longer available. this can happen if the
87      // group structure changes between the creation of the proposals
88      // and the call to this function.
89      return false
```

2.3.4 Group proposals

The following functions are used for deciding if a new group should be joined and how this actually happens. The function vote_PROPOSE_GROUP shown in lines $90-96$ decides locally if a node wants to join a given group or not. These votes are then communicated to all other nodes and using a consensus algorithm an agreement about whether to join or not to join the group is made. The result is either **COMMIT** or **ABORT**. After this every node calls decision_PROPOSE_GROUP shown in lines $98-110$ using the result of the consensus and possibly joins the group, and if joined marks the group data structure as temporary. After all nodes have executed decision_PROPOSE_GROUP group building is finalized. This is done by calling the function finalize_PROPOSE shown in lines $112-123$ with an additional argument which can be used to check if nodes have crashed **after** decision. Finalizing makes the group permanent and removes the temporary flag.

Listing 2.6: Utility functions

```
90  function vote_PROPOSE_GROUP(group)
91    // Find the member for which this node is a terminal node.
92    var mygid ← gid ∈ group.Members, ID ∈ gid
93    if want_to_join(mygid, group)
94      return COMMIT
95    else
96      return ABORT
97
```

```
98  function decision_PROPOSE_GROUP(result, group)
99    // Find the member for which this node is a terminal node.
100   // Find the member for which this node is a terminal node.
101   // For example if this node has the node address {3} and
102   // the group has the following members {{4,7,8},{9},{3,9,10}}
103   // the next line returns the group {3,9,10} because this node is
104   // a terminal node in this group.
105   var mygid ← {gid : gid ∈ group.Members ∧ ID ∈ gid}
106   var gid ← group.Terminals
107   if result = COMMIT
108     if want_to_join(gid, group)
109       join_group(gid, group)
110       Group[gid].LockedBy ← nbac_id
111
112 function finalize_PROPOSE(result, finalize_result, group)
113   var mygid ← {gid : gid ∈ group.Members ∧ ID ∈ gid}
114   var gid ← group.Terminals
115   if is_locally_consistent(group) ∧
116     Group[gid].LockedBy = nbac_id ∧
117     result = COMMIT
118     // Node crashed after decision
119     if finalize_result = ABORT
120       leave_group(gid)
121     // Group is now persistent on all nodes
122     if finalize_result = COMMIT
123       Group[gid].LockedBy = ⊥
```

2.3.5 Group checking

The functions shown in Listing 2.7 are used by the periodic group checking algorithm. If a node receives a request to check for groups shown in lines $124 - 131$, it first checks if the group still exists locally, i.e., it checks if the group has the same members and connections. If yes it returns the connections for which it is responsible as a piggy-back payload in the response. Responsible in this context means that the node v is an incident vertex for the connection $c = (v_1, v_2, \omega) \in E$ and $\mathrm{id}(v) = min(\mathrm{id}(v_1), \mathrm{id}(v_2))$. Therefore every node will receive all current connection weights and can therefore calculate the new group weight in line 140. Using this new weight group consistency checks are performed. This approach has the benefit that connection weights at the nodes are updated and movement of a node will trigger a reconstruction of the topology. Lines $142 - 143$ show the test where it is enforced that the parent group has a larger weight than the group in which this node is a terminal node. The second test shown in lines $146 - 148$ tests if the updated group weight exceeds the weight of the parent group if available.

Listing 2.7: Group checking

```
124 function vote_CHECK_GROUP(group)
125   // Find the member for which this node is a terminal node.
126   var mygid ← gid ∈ group.Members, ID ∈ gid
```

```
127   if is_locally_consistent(group)
128     return (COMMIT, (Group[mygid].Weight,
129       ∑_{(a,b,w)∈group.Connections ∧ID=min(id(a),id(b))} get_connection_weight(c)))
130   else
131     return ABORT
132
133 function decision_CHECK_GROUP(result, group, commit_data)
134   // Find the member for which this node is a terminal node.
135   var mygid ← gid ∈ group.Members, ID ∈ gid
136   var gid ← group.Terminals
137   if is_locally_consistent(group)
138     if result = COMMIT
139       // Update weight of the group
140       Group[gid] ← calculate_weight(commit_data, group)
141       // Test if weight consistency has been violated
142       if Group[mygid].Weight ≥ Group[gid].Weight
143         leave_group(mygid)
144       // Test if the current weight exceeds the weight of the parent
145       // group
146       else if Group[gid].ParentID ≠ ⊥∧
147         Group[gid].Weight ≥ Group[Group[gid].ParentID].Weight
148         leave_group(gid)
149     if result = ABORT
150       // Leave a broken group when at least one node voted ABORT
151       if Group[mygid] ≠ ⊥ ∧ Group[mygid].ParentID = gid
152       leave_group(gid)
```

2.3.6 Atomic Commitment

In a distributed system transactions usually involve several participants [Ray96] . A fundamental issue in these systems is to ensure the consistency of data which is the aim of the *transaction* concept. At the end of the transaction participants are required to enter a *commitment protocol* in order to **COMMIT** or **ABORT** it. This is typically done in two phases where in the first phase every participant votes **COMMIT** or **ABORT**. In the second phase all participants commit the transaction, i.e., make it permanent or undo the changes.

Definition 26 (Non-blocking Atomic Commitment (NBAC))**.** *[Ray96, p10] is the problem of ensuring that all correct participants of a transaction take the same decision which is either abort or commit. If the decision is **COMMIT** changes are made permanent and in case of **ABORT** no changes are made. A NBAC protocol protocol needs to fullfill the following properties.*

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 **COMMIT**.*

Non-Triviality: *If all participants vote **COMMIT** and there is no failure suspicion then the outcome of the decision is **COMMIT**.*

Our basic implementation of the NBAC protocol is shown in Listing 2.8 where its implementation is suited for an asynchronous system with reliable links. It is based on [Ray96, p14] with some modifications. A node wanting to initiate a NBAC instance with a set of *participants* calls the function initiate shown in Lines $1-5$. Whenever a node receives a NBAC message it executes the function 'nbac' and decides if it wants to commit or abort this transaction in Line 8 by calling the function 'vote_IMPL'[10]. The exact meaning of commit and abort is application dependent; in our case it is either wheter a group should be joined or wheter a group is consistent. After the decision it broadcasts its vote to all other participants in Line 9.

Now it waits for all votes to arrive or for a node being suspected in line 10. All currently available votes are collected in the mapping $vote : V \mapsto Votes$. The same concept is used for the predicate $suspected : V \mapsto \{true, false\}$ where the actual implementation is driven by a failure detector. If any node votes **ABORT** or a node was suspected the node calls the consensus algorithm with **ABORT**. In case all nodes voted **COMMIT** every node calls the consensus algorithm with **COMMIT** and the only allowed outcome of the consensus algorithm is **COMMIT**. A definition and the properties of a uniform consensus algorithm is given in Definition 27. Finally the piggy-packed commit data is extracted from all votes and stored in the variable *global_commit_data* in line 23. This additional data is then passed to the decision function in line 24 where the actual implementation dependent function 'decision_IMPL' handles it.

Listing 2.8: Non-blocking Atomic Commitment

```
1  function initiate(participants, data)
2    // Generate a unique NBAC id to allow parallel instances
3    var nbac_id ← unique_id( )
4    // Multicast to all participants including the initiator.
5    multicast(nbac_id, participants, data)
6
7  function nbac(nbac_id, participants, data)
8    (vote, commit_data) ← vote_IMPL(data)
9    multicast((vote, commit_data), participants)
10   wait for
11     // either if all votes have been received.
12     ∀v ∈ participants : vote(v) ≠ ⊥ ∨
13     // or a participant has been suspected
14     ∃v ∈ participants : suspected(v)
15
16   if ∃v ∈ participants : vote(v) = ABORT
17     result ← unified_consensus(ABORT)
18   if ∃v ∈ participants : suspected(v)
```

[10] IMPL stands for implementation and should indicate that this is application dependent.

```
19      result ← unified_consensus(ABORT)
20    if ∀v ∈ participants : vote(v) = COMMIT
21      result ← unified_consensus(ABORT)
22
23    var global_commit_data ← {commit_data : v ∈ participants ∧ (vote, commit_data) ∈ vote(v)}
24    decision_IMPL(nbac_id, result, data, global_commit_data)
```

The functions vote_IMPL and decision_IMPL are either instances of the group checking or group proposal functions described in the previous section. The uniform consensus algorithm used is the classic one where we will only repeat its definition given in [Ray96, p6].

Definition 27 (Uniform consensus)**.** *A* uniform consensus *algorithm ensures that all processes irrevocably decide on a common output value which is one of the input values. Uniformity adds the requirement that a correct process and a crashed process that decided just before crashing cannot decide differently. The algorithm has to fullfill the following properties.*

Termination: *Every correct process eventually decides on some value.*

Integrity: *A process decides at most once.*

Uniform agreement: *No two processes decide differently.*

Validity: *If a process decides on a value then this value must be one of the input values.*

The actual implementation requires an additional step for the group construction[11] which has been referred to as *finalization* by Thallner. The primary reason for Thallner to implement the finalize step was to postpone the group checking during construction [Tha05, p32]. For simplicity, Thallner used two NBAC instances where the first decision unlocked the voting of the second instance to synchronize nodes. This results in a high message complexity and also imposes implementation problems. Our implementation differs from that of Thallner in [Tha05] to avoid the usage of an additional NBAC but tries to match the original properties.

Definition 28. *The purpose of the* finalization *is to inform a node that* decision *has been executed on all other nodes [Tha05, p.32]. Finalization has the following properties:*

Local finalization: *Eventually finalize_IMPL will be called on each correct participant.*

Finalization agreement: *No two participants decide on different finalize results.*

Finalization validity: *If a participant decides on* $finalize_{result} = result = \boldsymbol{COMMIT}$ *then decision has terminated on every participant.*

Finalization non-triviality: *If there is no failure suspicion then* $finalize_{result} = \boldsymbol{COMMIT}$.

We will now present our modified algorithm in Listing 2.9.

[11] The NBAC shown in Listing 2.8 would be sufficient for the group checking.

Listing 2.9: Non-Blocking Atomic Commitment

```
1  function initiate(participants, data)
2    var nbac_id ← unique_id( )
3    // Multicast to all participants including the initiator.
4    multicast(nbac_id, participants, data)
5
6  function nbac(nbac_id, participants, data)
7    (vote, commit_data) ← vote_IMPL(data)
8    multicast((vote, commit_data), participants)
9    wait for
10     // either if all votes have been received.
11     ∀v ∈ participants : vote(v) ≠ ⊥ ∨
12     // or a participant has been suspected.
13     ∃v ∈ participants : suspected(v)
14
15   if ∃v ∈ participants : vote(v) = ABORT
16     result ← unified_consensus(ABORT)
17   if ∃v ∈ participants : suspected(v)
18     result ← unified_consensus(ABORT)
19   if ∀v ∈ participants : vote(v) = COMMIT
20     result ← unified_consensus(ABORT)
21
22   // this is only valid on commit
23   var global_commit_data ← {commit_data : v ∈ participants ∧ (vote, commit_data) ∈ vote(v)}
24   // locally call decision. this might happen at different times
25   // at any node.
26   decision_IMPL(nbac_id, result, data, global_commit_data)
27
28   // we have called decision. inform all other nodes about this.
29   multicast(FINALIZED, participants)
30
31   // wait for all nodes to finish their decision.
32   wait for
33     ∃v ∈ participants : suspected(v) ∨
34     ∀v ∈ participants : received(FINALIZED)
35
36   if ∃v ∈ participants : suspected(v)
37     finalize_IMPL(nbac_id, result, ABORT, data, global_commit_data)
38   if ∀v ∈ participants : received(FINALIZED)
39     finalize_IMPL(nbac_id, result, COMMIT, data, global_commit_data)
```

This implementation has the same properties as that of Thallner. The initiate function shown in lines $1-4$ is the same as in Listing 2.8. Lines $6-26$ also match the original NBAC.

Proof. The *Local finalization* property holds because if a node is correct it will multicast a FINALIZE message in line 29. The network is reliable and therefore if all nodes have sent the message every node will exit the wait statement in line 34. If a node is suspected it will eventually be detected and added to the set of suspected nodes. Therefore every

participant will eventually exit in line 33. □

Proof. Finalization validity holds because a node can only receive all FINALIZE messages if every participant has executed decision. A node will only call finalize_IMPL with **COMMIT** if it passes the test in line 38.

□

Proof. Finalization non-triviality holds because if no node is suspected the only way to exit the wait statement in line 34 is by the reception of all FINALIZE messages. And therefore the decision is **COMMIT** because of lines 38 and 39. □

Proof. Finalization agreement does only hold if we do not allow node crashes. Otherwise one node could be suspected and therefore two nodes might decide differently. If all nodes are correct then this holds for our modification. □

We do not see this a limitation at this stage of our implementation because solving consensus in a completely asynchronous system with even a single node failure is impossible [FLP85]. The usage of a consensus protocol with a failure detector was not practical for an implementation and therefore we require in the first version of our algorithm that no nodes crash during this time. If this assumption is violated, our simulation environment will report the violation. The second reason for this is that the network is unreliable and we implemented a reliable protocol on top of the network to allow a solution of the NBAC problem. Unfortunately, this reliable transmission using retransmits does not tolerate node failures.

What should also be questioned is why the finalization enforces such strong requirements because this is not obvious to the author. In a fully asynchronous system finalization is surely required for the correctness proof because if messages can be delayed arbitrarily long every proposed group can be destroyed by postponing one decision until the next group checking. A simple implementation could simple enforce that a group lives a certain amount of time such that we can assure that decision has finished at every node.

2.4 Propose Modules

A propose module is an independent and network specific part of the topology construction algorithm [TM05, p57]. Propose modules can be optimized for fast construction, low memory consumption, message size, topology convergence time, ... Because of their distributed nature, almost all propose modules use search strategies to explore the network. Different types of propose modules are classified according to the classification scheme introduced in Definitions 18, 19, 20, 21 and 22 shown in Section 2.2.1.

2.4.1 Network Model

Because the propose modules are triggered infinitely often during execution, it is not necessary to support any error handling. In case of an error a proposal is simply aborted and the worst thing which can happen is that some unnecessary messages have been sent which implies a waste of energy and time. We therefore assume the following network model for propose modules:

Unreliable links Message transmission is completely asynchronous and messages which are sent are only delivered with a certain probability $p > 0$.

Node crashes A node may crash fail-silent at any time during the execution.

Node recovery Late joining of nodes is allowed.

Not fully connected It is not required that the network is fully connected for correct execution. But of course a (reasonable) propose module will only form groups which are possible in the underlying transmission graph.

2.4.2 Supporting functions

In this section we will repeat the sub-functions required by our propose module, but with some modifications to the versions of Thallner shown in [TM05, p59-p63]. These modifications are required to allow their execution in non-fully connected networks. Furthermore some bugs have been discovered and the notation has been changed to an easier one. Important changes are highlighted and explained in more detail if necessary.

The function 'calculate_proposal' shown in Listing 2.10 takes a set of members called fix, which will be in the final group, and a set of *edges*, and computes the optimal group which can be built. In addition it supports an argument called min_weight which enforces that the weight of the calculated group is always bigger or equal to $min\_weight + \epsilon$. Finding the optimal group requires the availability of all edges between all terminal nodes, which is checked in line 20. If not all edges are available, the argument $allow\_missing$ can be used to allow a default weight[12]. The following modifications have been made to the original version shown in [TM05, p60]:

[12] Not all edges are available when the algorithm executes in a non fully connected network or if they could not be obtained because of various reasons (network problems, transmission errors, ...).

- The original version misbehaved when called with a single node group. This special case can happen for example in the local propose module. The reason for the misbehavior is that the function 'source' does not work correctly if $|fix| = 1$ because the test in line 306 in [TM05, p61] will never evaluate to true. Furthermore it would increment a in line 322 and would access an invalid array index in 326. The new version shown in Listing 2.10 takes care of this in the lines $10 - 12$.

- The original version did not set the *members* field of the group structure. This is essential for a correct group to be built. This has been added in line 7 in Listing 2.10.

- The original version supported obtaining missing edges during the calculation. This has been removed from the function and if required must now be done by the caller. The reason for this is that the new implementation uses a state machine based approach and storing the state is always easier at the beginning of the computation than during it[13].

- The function 'calculate_proposal' contains a problem if the calculated weight is smaller than the minimal group weight. The problem is that during the calculations the group weight might be passed as an argument to this function (see line 161 in Listing 2.15). If the group found after probing all possible combinations has exactly the same weight as the argument *min_weight* the original version of 'calculate_proposal' would return this value. The implementation of the group checking also needs to calculate the current weight of the group which would be $min\_weight + \epsilon$. If this happens groups would be built and destroyed rapidly which happened to the author during simulation and the algorithm did not converge. The reason for this is that the propose module would release a group which is ϵ better than the current existing group although it has the same members, terminals and connections.

Listing 2.10: Proposal calculation

```
1  // Global variable which holds the result during computation.
2  global P_out ← ⊥
3
4  function calculate_proposal( fix, min_weight, edges, allow_missing )
5    P_out.Connections ← ∅
6    P_out.Terminals ← ∅
7    P_out.Members = fix
8
9    // If called with |fix| = 1 we can immediately calculate the result
10   if |fix| = 1 then
11     P_out.Terminals ← {x : x ∈ fix} // fix is either {v} or {v_1, ... v_k}.
12     P_out.Weight ← 0
```

[13] The obvious reason behind this is that the internal state increases in complexity during computations and is a minimum at the beginning and at the end of it.

```
13    // Check for missing edges and abort if no default can be assumed.
14    else if |fix| > 1 then
15      P_out.Weight ← ∞
16      terminals ← {x : x ∈ x', x' ∈ fix}
17      completed_edges ← edges
18      // Check if we have all connections between all terminals
19      for x, y ∈ terminals
20        if x ≠ y ∧ (x, y, ω) ∉ edges
21          if allow_missing = false
22            // Edges missing. Signal error
23            signal error
24          else
25            completed_edges ← completed_edges ∪ {(x, y, ω_min)}
26
27    // Call the recursive algorithm to find the optimal edges
28    source(1, 1, ∅, 0.0, fix, completed_edges)
29
30    // Enforce a lower bound on the weight of this group
31    if P_out.Weight ≤ min_weight
32      P_out.Weight ← min_weight + ε
33
34    return P_out
```

The function source shown in Listing 2.11 builds the group internal connection by probing all possible combinations. The combinations are generated in the lines $53-56$ where the sequence starts with $< a, b >=< 1, 1 >$. For example let us assume that $k = 3$. Therefore we would eventually call the function 'calculate_proposal' with three groups and we would expect it to return the best possible group which can be built from it. The argument *min_group* would in this case be set to the minimum weight of any the groups. The function now starts to probe all possible connections between terminals. It starts by selecting a *source terminal* which is done by the function 'source' or if there are more than one it branches and tries the calculation for all of them. Then the function 'destination' tries to find a destination terminal. Again if there are more than one it branches. This algorithm executes recursively until all possible combinations has been probed.

To start let us assume that $|fix| = 3$ (for $k = 3$) and that $fix = \{\{v_1\}, \{v_2\}, \{v_3\}\}$. The first call would add a connection from $fix[1] = v_1$ in line 60 to $fix[2]$ in line 74. The second part is already part of the function 'destination' shown in Listing 2.12. This function again calls 'source'. Now b becomes 3 and a connection is created from $fix[1] = v_1$ to $fix[3] = v_3$. If source is called the next time a becomes 2 and b becomes 3 after the lines $53-56$ has been executed. This creates a connection between $fix[2]$ and $fix[3]$. Therefore we have now the set of connections $\{(v_1, v_2, \omega_{v_1,v_2}), (v_1, v_3, \omega_{v_1,v_3}), (v_2, v_3, \omega\, v_2, v_3)\}$. Because of the test in line 37 the lines $37-51$ are executed. The purpose of this line is to check if the found group is better than an already found one. Because this is the first (and only one) in this example it becomes the final result.
Next we have to calculate the set of terminal nodes. This is done in lines $42-51$. In case a member of fix is a single terminal node it must be in the final set because initially

it had k connections left and now has one left because of the internal connections. It is therefore added to the set of terminal nodes in line 45. If a member of fix is a group, it is of the form $\{v_1, \ldots, v_k\}$, the situation is somewhat different. Every terminal node must have a degree of $k-1$. If any node is an adjacent vertex in a connection, it will no longer be available as a terminal node because it will have degree k after construction. Therefore we simply iterate over all connections and remove nodes and finally one terminal node will remain which is the terminal node for this group. This is shown in lines $48-51$.

To see that the probing of all combinations also works correctly where a member of fix is a group consider the case $fix = \{\{v_1\}, \{v_2\}, \{v_{3_1}, v_{3_2}, v_{3_3}\}\}$. A textual description becomes quite complicated despite this simple example. Therefore Figure 2.10 shows the generated combinations where the nodes of the graph show the values of a and b and the currently added connections directly at the function entry point of source. At the end there are 9 groups to evaluate and the best one these will be chosen. Note that this algorithm produces quite a large number of combinations. The number of terminal nodes to consider are up to k^2, and we will always choose a subset of k internal connections. Therefore the algorithm is in $O\left(\binom{k^2}{k}\right)$; this amounts to $c \cdot 83$ for $k = 3$ and $c \cdot 1820$ for $k = 4$ where c is a constant. From a practical point of view, these results are not very satisfying, because for generating a proposal a lot of such calculations have to be done. Of course, considering complexity on n, this algorithm is in $O(1)$, but this result should be taken with caution.

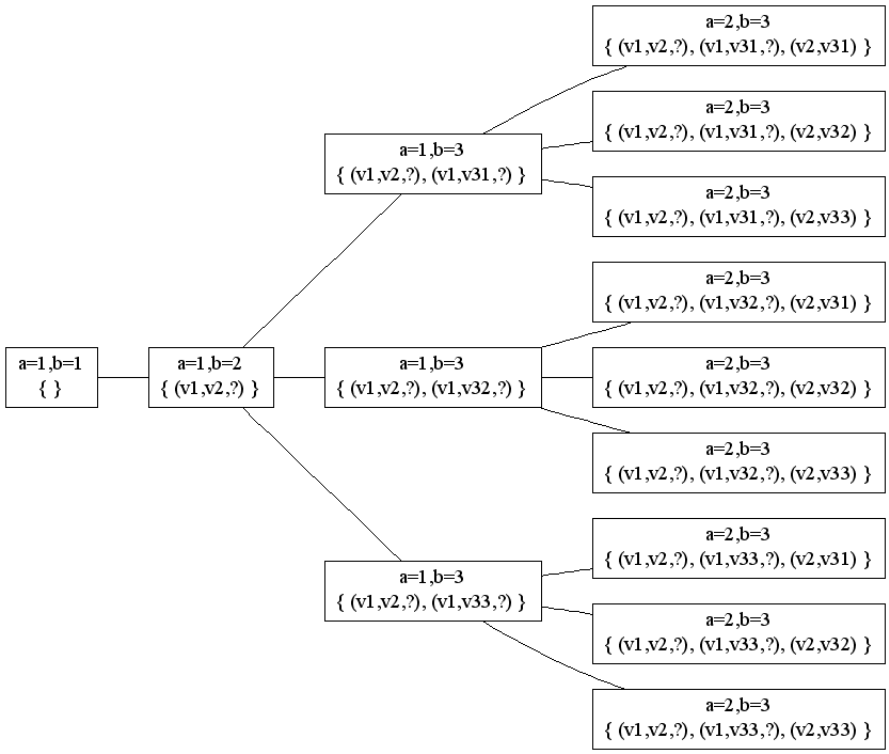


Figure 2.10: All combinations generated for the members $fix = \{\{v_1\}, \{v_2\}, \{v_{3_1}, v_{3_2}, v_{3_3}\}\}$

Listing 2.11: Proposal supporting function for source terminals

```
35 function source(a, b, connections, weight, fix, edges)
36
37   if a = |fix| − 1 ∧ b = |fix|
38     if P_out.Weight > weight
39       P_out.Weight ← weight
40       P_out.Connections ← connections
41       P_out.Terminals ← ⊥
42       for gid ∈ fix
43         if gid ∈ V
44           // gid is a single−node group
45           P_out.Terminals ← P_out.Terminals ∪ gid
46         else
47           // find the node in gid={v_1, ..., v_k} with one connection left
48           terminal ← gid
49           for c = (x, y, ω) ∈ connections
50             terminal ← terminal \ {x, y}
51           P_out.Terminals ← P_out.Terminals ∪ terminal
52   else
53     if b = |fix|
54       a ← a + 1
55       b ← a
56     b ← b + 1
57
58     if fix[a] ∈ V
59       // for a single node there is no choice for a connection.
60       c.x ← fix[a]
61       destination(a, b, connections, weight, fix, edges, c)
62     else
63       available_terminals ← fix[a]
64       // remove all terminal nodes which already have a connection.
65       for i ∈ connections
66         available_terminals ← available_terminals \ {c.x ∪ c.y}
67       // probe all remaining terminals.
68       for gid ∈ available_terminals
69         c.x ← gid
70         destination(a, b, connections, weight, fix, edges, c)
```

We already used the function 'destination' before. It is used to probe the destination terminals and is shown in Listing 2.12. If the member $fix[b]$, to which a connection should be built, is a single node group then there is no choice left and a connection is made in line 74. In the other case multiple choices are possible. We start by calculating the set of available remaining terminals in lines $79 - 82$. Initially this set has a size of k, but during probing its cardinality might decrease because connections are created and terminal nodes become "normal" nodes. Every possible destination is probed which can be seen by the branch of calls to 'source in line 87. In any case, a new connection is added to the set *connections* and the group weight is adapted. It is important to note that the weight restrictions from Thallner can be enforced by this implementation by

using the additional argument *min_weight* of the function source shown in Listing 2.11.

Listing 2.12: Proposal supporting function for destination terminals

```
71 function destination(a, b, connections, weight, fix, edges, c)
72   if fix[b] ∈ V
73     // create connection.
74     c.y ← fix[b]
75     c.weight ← ω : (c.x, c.y, ω) ∈ edges
76     source(a, b, connections ∪ c, weight + c.weight, fix, edges)
77   else
78     // probe all possible destination terminals.
79     available_terminals ← fix[b]
80     // remove all terminal nodes which already have a connection.
81     for c ∈ connections
82       available_terminals ← available_terminals \ {c.x ∪ c.y}
83     // probe all remaining terminals
84     for gid ∈ available_terminals
85       c.y ← gid
86       c.weight ← ω : (c.x, c.y, ω) ∈ edges
87       source(a, b, connections ∪ c, weight + c.weight, fix, edges)
```

Another basic function is the function potential_member_set shown in Listing 2.13, which takes an array of members fix, which must be in the final group, and a set of potential members, and calculates the potential members which can be used to build a group of $|fix|+1$ members. Restrictions which cause the final set *pms* to be smaller than the set *ps* are for example the specified *bound*, that some connections are missing, or that a group cannot be built because of consistency issues, for example if two members would share the same terminal node. To give a concrete example let us assume that $fix = \{\{1,2,3\},\{5\}\}$, $ps = \{\{4\},\{6\}\}$, and the bound is 100. The algorithm would call calculate_proposal for the groups $\{\{1,2,3\},\{5\},\{4\}\}$ and $\{\{1,2,3\},\{5\},\{4\}\}$ in lines $101-102$, and would add them to the set *pms* in line 106 if the weight bound is not exceeded. Finally, the set of potential members is checked for consistency in lines $111-114$ to ensure that a group of size k can actually be built. In our example all groups would pass because they are valid. Somewhat unclear to the author of this work is the usage of the $filter$ argument and the actual implementation in lines $91-95$. Furthermore we had to add additional checks because contrary to Thallner we do not require that there is a connection between every node, at least not in the propose modules. Therefore we try to continue building proposals with the check shown in line 103 even if a single calculation failed.

Listing 2.13: Potential member set calculation

```
88 function potential_member_set(fix, ps, edges, bound, filter)
89   pms ← ⊥
90
91   if filter
92     if fix ∩ V ≠ ∅
93       for i ∈ ps
```

```
94              if (i ∈ V) ∧ (id(i) < min(id(fix) ∩ id(V)))
95                ps ← ps \ i
96
97      // remove members in ps which are already in fix
98      ps ← ps \{x : x' = y' ∧ x' ∈ x, x ∈ ps, y' ∈ y, y ∈ fix}
99
100     // build groups of |fix|+1 members and check for bound
101     for i ∈ ps
102       P' ← calculate_proposal(fix ∪ {i}, ∅, edges, false)
103       if error signaled
104         pms ← pms ∪ {i}
105       else if P'.Weight ≤ bound
106         pms ← pms ∪ {i}
107
108     // remove candidates which can not be used to build groups. note t
109     // hat k − |fix| − 1/ groups are required to build a complete
110     // group.
111     if k − |fix| > 1
112       for i ∈ pms
113         if |potential_member_set(fix ∪ {i}, pms \{i}, edges, bound, false) < k − |fix| − 1
114           pms ← pms{i}
```

Finally we present some supporting functions in Listing 2.14 with are very basic and do not require further explanation.

Listing 2.14: Supporting function used by the propose modules

```
116 function leader_of(gid)
117   return min({x : x ∈ gid})
118
119 function is_consistent(members)
120  // a set remove duplicates
121   nodes ← {x : x ∈ x', x' ∈ members}
122   count ← 0
123   for gid ∈ members
124     count ← count + |gid|
125   // if one node is used in different groups the sizes differ
126   return count = |nodes|
```

2.4.3 Local Non-Perfect Propose Module

We will only describe the *Local non-perfect propose module*, briefly LNP propose module. We have made some corrections and adjustments to the version of Thallner introduced in [TM05, p68] to fix some problems and solve some implementation issues. The actual implementation is shown in Listing 2.15.

- Missing edges are now fetched prior to calling 'calculate_proposal', because it makes the implementation simpler. The reason for this is that obtaining edges must be done by exchanging messages, and therefore the edges are not immediately

available. Because we have to follow a polling based approach, we wanted to reduce the state to store and therefore moving this to the beginning seemed to be an obvious and good approach.

- A group is only forwarded to its parent group in line 469 if the new group is consistent. Inconsistent in this sense means that it can never form a valid group, i.e., when two members share the same terminal nodes. These modifications can be found in Listing 2.15 in lines $196 - 197$.

- We added an additional check in line 156 to check if the group has been destroyed in the meantime. This happens when a proposal is forwarded and in the time window between the forwarding and the delivery of the search message the group *gid* is destroyed.

- The relevant edges are extended with the connections needed to calculate the weight of the new group in line 185. Contrary to Thallner in line $463-464$ we do not only add connections which are adjacent to this node but also required connections for the extended set of fix members. This is only a best effort implementation and if not available the leader receiving the search must perform the queries anyway.

We will write $arr[i], i \in \mathbb{N}$, to access the element i of the array arr where the index starts at 1. An element in the array is created by assigning it a value using the notation $arr[i] \leftarrow x$. An element is removed from the array by assigning it the undefined value $\perp$, written as $arr[i] \leftarrow \perp$. We write $arr.Length$ to access the number of currently assigned items in this array.

The algorithm itself follows a DFS approach. A node which is currently in charge of the SEARCH message tries to calculate a new proposal in line 161 for the members fix already found during the search and the group identifier for which it is a leader. To calculate this proposal, we first have to fetch any missing edges in lines $151 - 152$. After the proposal has been calculated, we check if the parent groups of all members have higher weights than the weight of the calculated group. The parent group for *gid* is directly accessible because this node is a leader for this group and it is stored in the variable *pgid*. The bound used is then the minimum weight of the parent group *pgid* or the minimum weight of any of the group members. This calculation is performed in lines $163 - 166$.

Now there are two possibilities. If the bound is not violated, this group looks promising and we continue in lines $169 - 190$. If we already have k members, which is tested in lines 170, we are done and we send the search back to the originator[14]. If we have less than k members, we need to find additional members for the group. We calculate the set of potential members in line 175, and if there are enough members left to build a complete group, we continue after line 176. Next we look for our best neighbor in lines $178 - 180$ and if there is a neighbor available we build a new search message with ourself as member in lines $182 - 187$. The addition of the current *gid* to the *excluded* set is

[14] It it unclear to the author why this is required because this node could also initiate the group proposal because any terminal nodes must participate.

important because it prevents nodes/groups from being visited twice. The actual forwarding (of the still incomplete group) is done in line 189.
The other case is when the group weight does not allow us to build a group. That is the weight of the group is higher than any of its members parent groups. The first case is when there exists a parent group. This is checked in line 195. If a new group is theoretically possible with *pgid* as a member instead of *gid*, which is checked in line 196, then the SEARCH is forwarded to the leader of the parent group in line 196. Otherwise we cut this search path by popping the last elements from our DFS search in lines $203-207$, and by adding the group *gid* to the excluded set. The search is then sent back to the previous leader which sent us this search message in line 211.
Some small things, are still worth mentioning, which we skipped during the explanation. The variable *min_weight* is used to enforce that groups built have higher group weights than the current group. Therefore a calculated group weight must be within the upper bound *bound* given by the parents of its members and the minimum group weight enforced by the weight of the members themselves. If a result is received by a node in line 213, it is released to the group creation algorithm by calling the function release_proposal.

Listing 2.15: local_non_perfect_proposal

```
128 function local_non_perfect_proposal(gid)
129   send(SEARCH, ∅, {V}, {gid}, {∞}, ∅, gid) to leader_of(gid)
130
131 function calculate_required_edge_set(groups)
132   // calculate the required edges to probe all group internal
133   // connections.
134   terminals = {x|x ∈ x', x' ∈ groups}
135   edges = (terminals × terminals) \{(x, x) : x ∈ terminals}
136   return edges
137
138 function complete_edge_set(edges, groups)
139   terminals = {x|x ∈ x', x' ∈ groups}
140   all_edges = (terminals × terminals) \{(x, x) : x ∈ terminals}
141   // E is the node local edge set
142   return edges ∪ (all_edges ∩ E)
143
144 if received(SEARCH, fix, ps, excluded, bound, min_weight, edges, gid)
145
146   completed_edges ← edges
147   // check if some edges are missing in the set edges we have
148   // already received.
149   missing_edges ← \\
150     calculate_required_edge_set(fix ∪ {gid}) \ {(x, y) : (x, y, ω) ∈ edges}
151   if missing_edges ≠ ∅
152     completed_edges = fetch_missing_edges(missing_edges)
153
154   // check if this group still exists. this can happen if it
155   // has been destroyed during the search
```

```
156   if Group[gid] ≠ ⊥
157     pgid ← Group[gid].ParentID
158     depth ← |fix| // depth of DFS search
159
160     P' ← calculate_proposal(fix[depth] ∪ {gid}, max(min_weight, Group[gid].Weight), \\
161       completed_edges)
162
163     bound2 ← bound[depth]
164     if pgid ≠ ⊥
165       bound2 ← min(bound2, Group[pgid].Weight)
166     bound2 ← bound[depth]
167
168     finished ← false
169     if P'.Weight < bound2
170       if |P'.Members| = k // send back final result
171         send(RESULT, P') to_leader_off(fix[depth][1])
172         finished ← true
173       else if |P'.Members| < k
174         pms ← potential_member_set(fix[depth] ∪ {gid}, ps[depth] \(excluded ∪ {gid}), \\
175           edges, bound2)
176         if |fix[depth] + 1 + |pms|| ≥ k
177           // calculate the potential connections, briefly called pc.
178           pc ← {(x, y, ω) : (x, y, ω) ∈ E ∧ x = ID ∧ y ∈ {z ∈ z', z' ∈ pms}}
179           // take the nearest neighbor from the set.
180           selected_neighbor ← {y : (x, y, ω) ∈ pc ∧ ω = min_ω({ω : (x, y, ω) ∈ pc}}
181           if selected_neighbor ≠ ∅
182             fix[depth + 1] ← fix[depth] ∪ {gid}
183             ps[depth + 1] ← pms
184             excluded ← excluded ∪ {gid}
185             edges ← complete_edge_set(edges, fix[depth + 1] ∪ {selected_neighbor})
186             bound[depth + 1] ← bound2
187             min_weight[depth + 1] ← Group[gid].Weight
188             send(SEARCH, fix, ps, excluded, bound, edges, {selected_neighbor}) to \\
189               selected_neighbor
190             finished ← true
191         else
192           // there are no more potential members. we abort the search.
193           finished = true
194
195     if finished = false
196       if pgid ≠ ⊥ ∧ is_consistent(fix[depth] ∪ Group[pgid])
197         send(SEARCH, fix, ps, excluded ∪ {gid}, bound, edges, pgid) to leader_of(pgid)
198       else
199         if depth > 1
200           fix2 ← fix[depth]
201
202           // remove last DFS entry
203           fix[depth] ← ⊥
204           ps[depth] ← ⊥
```

```
205             excluded ← excluded ∪ {gid}
206             bound[depth] ← ⊥
207             min_weight[depth] ← ⊥
208             // send SEARCH back to originator with ourself in the
209             // excluded set.
210             send (SEARCH, fix, ps, excluded, bound, edges, fix2[|fix2|]) \\
211               to leader_of(fix2[|fix2|])
212
213 if received(RESULT, P')
214   // release proposal such that it can be processed in an NBAC
215   // instance to form a new group.
216   release_proposal(P')
```

The Thallner algorithm requires that nodes periodically generate proposals [Tha05, p17]. Looking at the algorithm we see that the local non-perfect propose module always includes its own group identifier [Tha05, p60]. To explore the full search space a node must try to find proposals for all groups in which the node is a leader. The pseudo code for this is shown in Listing 2.16. Every node starts by using its own node identifier shown in line 219. It then recursively climbs up the group hierarchy in line 223. If a node is a leader for the group, it initiates a local proposal containing this group shown in lines 221–222. Note that an actual implementation must use a threshold, and the proposals should be generated in a round-robin fashion because otherwise multiple proposal searches would be emitted to the network, which would lead to network congestion.

Listing 2.16: local_non_perfect_proposal

```
218 function propose_groups( )
219   cur_gid ← ID
220   while cur_gid ≠ ⊥
221     if leader_of(cur_gid) = ID
222       local_non_perfect_proposal(cur_gid)
223     cur_gid ← Groups[cur_gid].ParentID
```

Example

We will show an example how the implementation of the algorithm works in the network shown in Figure 2.11. Because of the high parallelism and to keep the example simple, we allowed only node 3 to start triggering a proposal. The first group which will be built is $\{3, 4, 7\}$.

1. At the start, the function propose_groups in Listing 2.16 is called at node 3. The variable *cur_gid* is set to 3 at the function entry point. We assume that no groups have been built, and therefore the algorithm only executes local_non_perfect_proposal ({3}) in line 222.

2. The function local_non_perfect_proposal is executed at node 3. We will write array elements enclosed in brackets and sets as usual in curly braces. Initially the

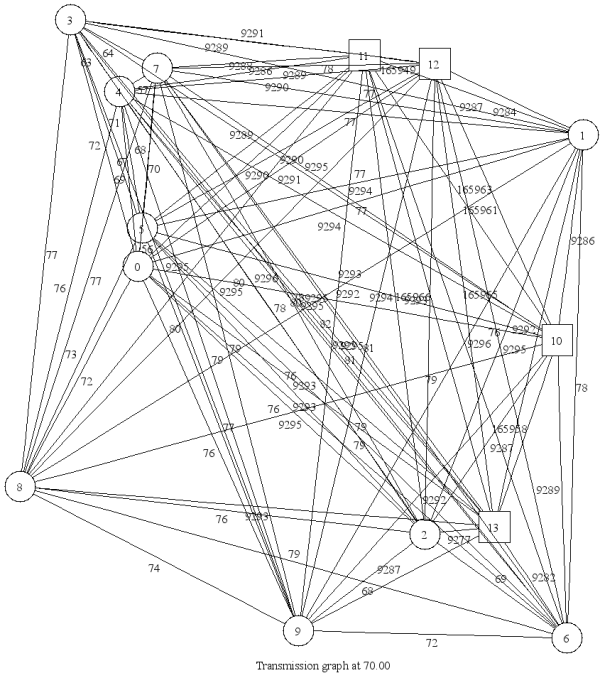


Figure 2.11: Example transmission graph for a network with $'n = 10$ and $n'' = 4$

variables are set to

$$\begin{aligned} fix[0] &= \{\} & ps[0] &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\\ & & & \{7\},\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\} \\ excluded &= \{\{3\}\} \\ bound &= [\infty] \\ min\_weight &= [0] \\ edges &= \{\} \end{aligned}$$

3. Now node 3 calculates a proposal in line 161. The calculated proposal $(\{\{3\}\},\{3\},\{\},0)$ is not complete, because it does not contain $k = 3$ members which is tested in line 170. It is finally forwarded to node 4, the best neighbor for node 3, in line 189.

4. Now node 4 receives the proposal, which has now a depth of 2 because fix now contains two arrays. The function arguments of received in line 144 now have the

values show below and *gid* equals {4}.

$$
\begin{aligned}
fix[0] &= \{\} & ps[0] &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\\ &&&\{7\},\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\}\\
fix[1] &= \{\{3\}\} & ps[1] &= \{\{0\},\{1\},\{2\},\{4\},\{5\},\{6\},\{7\},\\ &&&\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\}\\
excluded &= \{\{3\}\}\\
bound &= [\infty,\infty]\\
min\_weight &= [0,0]\\
edges &= \{\}
\end{aligned}
$$

Now node 4 fetches the missing edge, which is $(3,4,63)$ and calculates a proposal in line 161. The calculated proposal is $(\{\{3\},\{4\}\},\{3,4\},\{(3,4,63)\},63)$ This is still not sufficient to build a group, and therefore the proposal is forwarded again. The best neighbor for node 4 is 7.

5. Finally node 7 receives the search from node 4 with *gid* equaling {7} and variable values as shown below.

$$
\begin{aligned}
fix[0] &= \{\} & ps[0] &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\\ &&&\{7\},\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\}\\
fix[1] &= \{\{3\}\} & ps[1] &= \{\{0\},\{1\},\{2\},\{4\},\{5\},\{6\},\{7\},\\ &&&\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\}\\
fix[2] &= \{\{3\},\{4\}\} & ps[2] &= \{\{0\},\{1\},\{2\},\{5\},\{6\},\{7\},\{8\},\\ &&&\{9\},\{10\},\{11\},\{12\},\{13\}\}\\
excluded &= \{\{3\},\{4\}\}\\
bound &= [\infty,\infty,\infty]\\
min\_weight &= [0,0,0]\\
edges &= \{\}
\end{aligned}
$$

Together with node 7, the required number of members for $k = 3$ has been reached. The calculated proposal by node 7 is
$(\{\{3\},\{4\},\{7\}\},\{3,4,7\},\{(3,4,63),(3,7,64),(4,7,57)\},184)$.

6. This final proposal is then released to the group construction algorithm as, described in Section 2.3.4.

After the main algorithm has finished, and all members of the group have agreed, the connections are created, and the topology graph looks like the one shown in Figure 2.12.

What happens next is that the proposal search algorithm of node 3 is triggered again by a periodic timer, and 'propose_groups' is executed again. Again node 3 recursively walks up the group hierarchy, and now also initiates a search containing the group $\{3,4,7\}$. The execution of this search is shown below, although with a bit less detail as in the previous example.

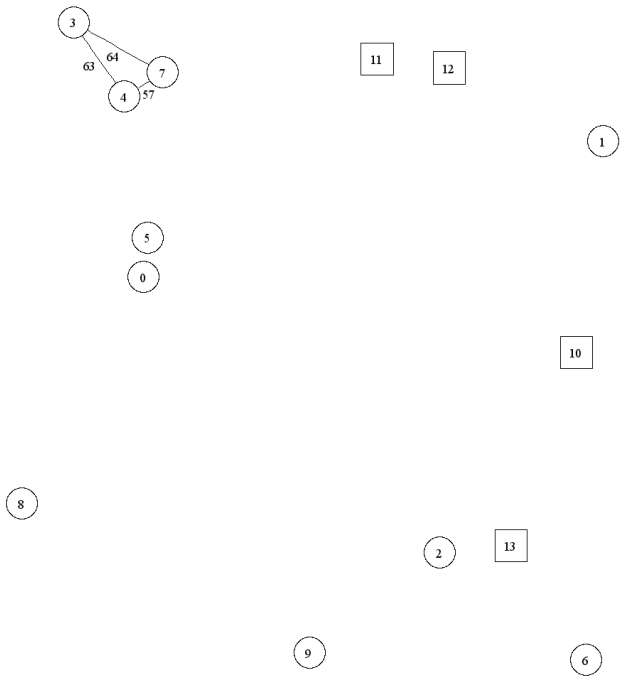


Figure 2.12: Overlay graph after first group has been built by the LNP propose module

1. The function local_non_perfect_proposal is executed using $\{3, 4, 7\}$ as an argument. Then the function 'received' is called, where the variables have the following initial values.

$$
\begin{aligned}
fix[0] &= \{\} \qquad & ps[0] &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\\
& & & \{7\},\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\}\\
excluded &= \{\{3,4,7\}\}\\
bound &= [\infty]\\
min\_weight &= [0]\\
edges &= \{\}
\end{aligned}
$$

Now node 0 again starts to calculate a proposal in line 161 but this time with a bound of 184 because this is the current weight of the group $\{3, 4, 7\}$. Because it is called the first time, teh group is incomplete, and again the set of potential members is calculated in line 175. The nodes $3, 4$ and 7 are already used, and the next best node is node 5, to whichthe search is forwarded.

2. Now node 5 receives the search from node 3 with the variables set to

$$\begin{aligned} fix[0] &= \{\} & ps[0] &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\\ & & & \{7\},\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\} \\ fix[1] &= \{\{3,4,7\}\} & ps[1] &= \{\{0\},\{1\},\{2\},\{5\},\{6\},\{8\},\{9\},\\ & & & \{10\},\{11\},\{12\},\{13\}\} \\ excluded &= \{\{3,4,7\}\} \\ bound &= [\infty,\infty] \\ min\_weight &= [0,184] \\ edges &= \{\} \end{aligned}$$

   There are still not enough members and after calculating a group proposal and the set of potential members the search is forwarded to node 0.

3. Finally node 0 receives the search from node 5 with $gid = \{0\}$ and the following variables.

$$\begin{aligned} fix[0] &= \{\} & ps[0] &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\\ & & & \{7\},\{8\},\{9\},\{10\},\{11\},\{12\},\{13\}\} \\ fix[1] &= \{\{3,4,7\}\} & ps[1] &= \{\{0\},\{1\},\{2\},\{5\},\{6\},\{8\},\{9\},\\ & & & \{10\},\{11\},\{12\},\{13\}\} \\ fix[2] &= \{\{3,4,7\},\{5\}\} & ps[2] &= \{\{0\},\{1\},\{2\},\{6\},\{8\},\{9\},\{10\},\\ & & & \{11\},\{12\},\{13\}\} \\ excluded &= \{\{3,4,7\},\{5\}\} \\ bound &= [\infty,\infty,\infty] \\ min\_weight &= [0,184,0] \\ edges &= \{\} \end{aligned}$$

   It calculates a new proposal for the group $\{0,3,5\}$ and releases the group proposal $(\{\{0\},\{3,4,7\},\{5\}\},\{0,3,5\},\{(0,5,56),(0,4,69),(5,7,68)\},193)$.

Again this group proposal is released to the main group construction algorithm and the final overlay graph is shown in Figure 2.13.

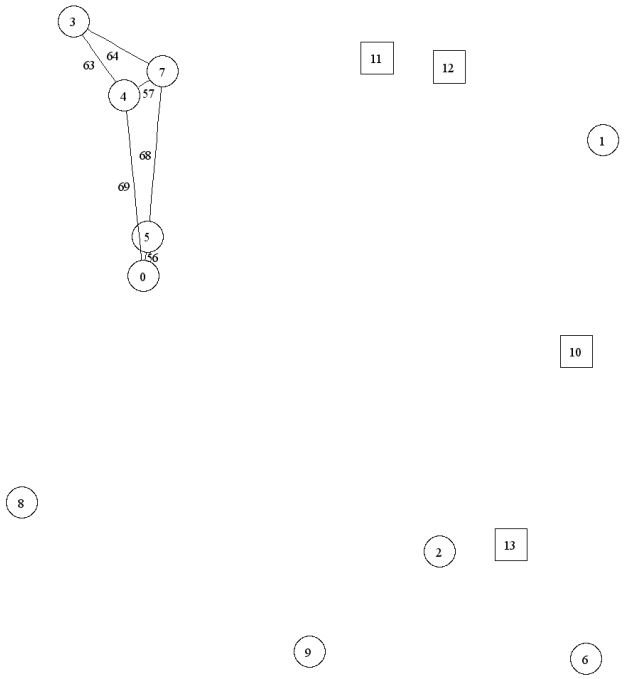


Figure 2.13: Overlay graph after second group has been built by the LNP propose module.

3 NS2

3.1 Introduction

3.1.1 NS2

NS2 is a discrete event simulator targeted at network research [FV06]. It provides support for simulation of different networks protocols like TCP and UDP as well as routing and multicast protocols over wired and wireless networks. We have chosen to use the wireless model of NS2 to implement our topology management algorithm, because it is the most widely used and respected network simulator in the scientific community.

NS2 is written in C++ and TCL, where TCL is used for scripting and C++ is used for the core implementation. Using a scripting language for the simulator configuration allows short development cycles, because no recompilation is required. The core was written in C++ because of performance reasons. The glue between C++ and TCL is TclCL (Tcl with classes) and OTcl (MIT Object TCL). All of our components are written in C++, with the exception of the node configuration. In NS2, a typical wireless node corresponds to Figure 3.1 where each building block corresponds to an object [FV06, p146].

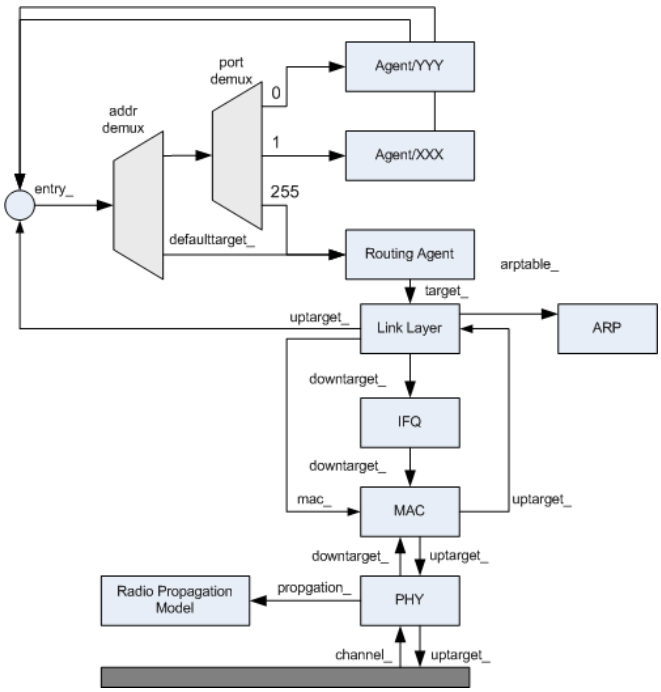


Figure 3.1: NS2 wireless model

Packets are passed between these objects using so called *targets*, where a target is simply a name for a special reference or a pointer to an NS2 object which can receive and/or send messages. These targets are setup by the node configuration interface, and if tracing is enabled, a trace object might sit between each target to record data packets passing between such instances. We will explain the most import objects briefly. The

interested reader is referred to the excellent NS2 documentation in [FV06]. We start from the bottom to the top in Figure 3.1.

PHY: The *PHY* is the physical interface. In our case, this is always set to Phy/WirelessPhy, which is the shared wireless media. This class is used by the mobile node to access the channel. It is subject to collisions and the radio propagation model and receives packets from other wireless nodes. The implementation tags each sent packet with "metadata" information like transmission power and wavelength. This information is used by a receiving node to decide if the packet can be received, captured or detected. For more information on this see [FV06, p.151].

MAC: Provides an implementation of the IEEE 802.11 MAC Protocol. It uses the classic RTC/CTS/ACK and DATA frames for unicast packets and simple sends out DATA for all broadcast packets. It is implemented in the class Mac/802_11 [FV06, p.151]. We will always use Mac/802_11 for our simulations.

IFQ: The interface queue in all our simulations is a priority queue implemented in Queue/DropTail/PriQueue. It is possible to priorize specific packets in this queue to improve performance. Although we did not implement it, it would make a lot of sense to priorize ACK packets used by the reliable multicast layer to prevent ACK explosion. For some hints on implementing this feature, see [FV06, p.151] and [FV06, p.69-p.83]. This problem is discussed in more detail in Section 3.6.2.

Link Layer: The link layer is used by data link protocols. Such protocols can implement fragmentation, packet reassembly, and reliable link protocols. Furthermore it has an associated ARP Module used to convert IP-addresses into MAC addresses and, if required, vice-versa. Outgoing packets are typically passed down from the *Routing Agent* to the link layer. Incoming packets are passed to the node *entry* point where they are dispatched.

Address Demux: The basic address classifier is used for unicast forwarding. It uses the packet address and applies a bitwise mask and shift operation to get a slot number. If the destination address is registered, it is passed to the local port demultiplexer. Otherwise it is passed to the default target, which is typically a routing agent to forward the packet.

Port Demux: The port demultiplexer is used by the agents. Every agent is assigned a new number on the port demultiplexer, and if two agents are connected, this information together with the address uniquely determines the flow through the network. This is explained in more detail later in this chapter.

Agent: Agents represent endpoints where network packets are consumed or created [FV06, p.95]. There are a lot of different protocol agents, for example Agent/TCP and Agent/TCPSink. On top of these transport agents, so called *Applications* can be used. Example applications are Application/FTP or Application/Traffic/Exponential. For a complete list, see [FV06, p.328-p.337]. An example is shown later in this chapter in Figure 3.2.

Routing Agent: If a packet is received by a node and it can not be handled local it is forwarded to the routing agent. A very basic routing agent is Agent/DumbAgent which performs no routing at all and drops non local packets. A more advanced routing agent is Agent/DSDV.

Radio Propagation Model: The model is used to predict the received signal power level of each packet [FV06, p.187]. At the physical level, every wireless node has a receiver threshold. If the signal power level is too low, the packet is marked and dropped by the MAC layer. Three readio propagation models are available in NS2. The *Free space* model assumes ideal propagation assuming line-of-sight. The default model we use is a *Two-ray ground reflection* model. This model considers the direct path and a ground reflection path between nodes [FV06, p187.]. The most advanced model available is the *Shadowing model*. It consists of two parts, a deterministic part for the path loss, and a log-normal random variable. Therefore, nodes can only probabilistically communicate when they are near the borders of their communication range [FV06, p.188].

To give a more concrete example, we will now show a very basic wireless setup where two nodes communicate with each other using a TCP connection with an FTP application on top of it. This example is one of the most basic examples in a wireless setup one can produce, yet it allows us to show the basic operation principles of the simulator. The setup script for the simulator is shown in Listing 3.1. Lines $18 - 75$ cover the basic wireless setup, and they do not need to be changed at all. In lines $80 - 88$, two wireless nodes are created and their position in the Euclidean plane is set. Furthermore, node movement is disabled in this setup. More interesting are lines $95 - 100$, where a TCP connection is established between node 0 and node 1. First a TCP agent is created in line 95 and attached to the node in line 96. In lines $97 - 98$, a FTP application is instantiated and attached to the agent. The FTP application will start to produce one packet at time 10.0, as configured in line 100.

Listing 3.1: Example setup script for TCP communication in NS2

```
1  # ----------------------------------------------------------------------
2  # Project: Simple example for two nodes communicating over a wireless
3  #   media.
4  # Author:  Christian Walter <e0225458@student.tuwien.ac.at>
5  #
6  # $Log: ns2-introduction-example-tcp-2-nodes-adhoc.tcl,v $
7  # Revision 1.2  2007/10/10 19:49:00  cwalter
8  #  - Final updates.
9  #
10 # Revision 1.1  2007/07/29 21:36:33  cwalter
11 #  - New code examples for ns2 chapter.
12 #
13 # Revision 1.1  2007/07/28 08:03:21  cwalter
14 #  - Added example script for TCP connection between two wireless nodes.
15 #
16 # ----------------------------------------------------------------------
17
```

```
18
19 # ---------------------------------------------------------------------------
20 # set configuration properties
21 # ---------------------------------------------------------------------------
22 set num_nodes           2                              ;# total nodes
23 set xsize               100
24 set ysize               100
25
26 set val(chan)           Channel/WirelessChannel        ;# Channel Type
27 set val(prop)           Propagation/TwoRayGround       ;# radio-propagation model
28 set val(netif)          Phy/WirelessPhy                ;# network interface type
29 set val(mac)            Mac/802_11
30 set val(ifq)            Queue/DropTail/PriQueue        ;# interface queue type
31 set val(ll)             LL                             ;# link layer type
32 set val(ant)            Antenna/OmniAntenna            ;# antenna model
33 set val(ifqlen)         100                            ;# max packet in ifq
34 set val(rp)             DumbAgent
35
36 # ---------------------------------------------------------------------------
37 # Create a simulator instance
38 # ---------------------------------------------------------------------------
39 set ns [ new Simulator ]
40 set tracefd [ open adhoc.tr w ]
41 $ns use-newtrace
42 $ns trace-all $tracefd
43
44 set topo [ new Topography ]
45 $topo load_flatgrid $xsize $ysize
46
47 set god_ [ create-god $num_nodes ]
48
49 set chan_1_ [new $val(chan)]
50 $ns node-config \
51     -adhocRouting $val(rp) \
52     -llType $val(ll) \
53     -macType $val(mac) \
54     -phyType $val(netif) \
55     -ifqType $val(ifq) \
56     -ifqLen $val(ifqlen) \
57     -antType $val(ant) \
58     -propType $val(prop) \
59     -topoInstance $topo \
60     -agentTrace ON \
61     -routerTrace OFF \
62     -macTrace OFF  \
63     -movementTrace OFF \
64     -channel $chan_1_
65
66 # ---------------------------------------------------------------------------
67 # Common functions
68 # ---------------------------------------------------------------------------
69 proc finish {} {
70     global ns tracefd
71     $ns flush-trace
```

```
72      close $tracefd
73      exit 0
74  }
75
76
77  #---------------------------------------------------------------------
78  # Create nodes
79  #---------------------------------------------------------------------
80
81  for { set i 0 } { $i < $num_nodes } { incr i } {
82    set node_($i) [ $ns node ] ;
83    $node_($i) random-motion 0
84  }
85
86  $node_(0) set X_ 10.0; $node_(0) set Y_ 15.0; $node_(0) set Z_ 0.0;
87  $node_(1) set X_ 60.0; $node_(1) set Y_ 75.0; $node_(1) set Z_ 0.0;
88
89
90  #---------------------------------------------------------------------
91  # TCP connection between node_(0) and node_(1)
92  #---------------------------------------------------------------------
93
94  # Create a TCP agent and attach an application two it.
95  set source [new Agent/TCP]
96  $ns attach-agent $node_(0) $source
97  set ftp [new Application/FTP]
98  $ftp attach-agent $source
99  # Start transfers at time = 10.0s and produce 1 packet
100 $ns at 10.0 "$ftp produce 1" set source [new Agent/TCP]
101
102 # The sink receives the traffic
103 set sink [new Agent/TCPSink]
104 $ns attach-agent $node_(1) $sink
105
106 # Connect the source and the sink, i.e. establish the connection.
107 $ns connect $source $sink
108
109
110 #---------------------------------------------------------------------
111 # Time schedule
112 #---------------------------------------------------------------------
113
114 $ns at 20.0 "finish"
115 $ns run
116
```

Following our graphical representation of a node object in NS2, this setup will result in the structure shown in Figure 3.2, where the low level physical details have been abstracted. An important thing to notice is that a connection is uniquely identified by the combination of the source address and the source port as well as the destination address and the destination port. The addressing scheme can also be seen in the trace file produced by NS2 shown in Listing 3.2. For example, in line 1 we can see that node 0 (−Ni 0) sends a packet from source address 0 and source port 0 (−Is 0.0) to the

destination address 1 with destination port 0 (−Id 1.0). In line 2, node 1 receives the packet and answers with an ACK in line 3. More information on the trace file format can be found in Appendix E.

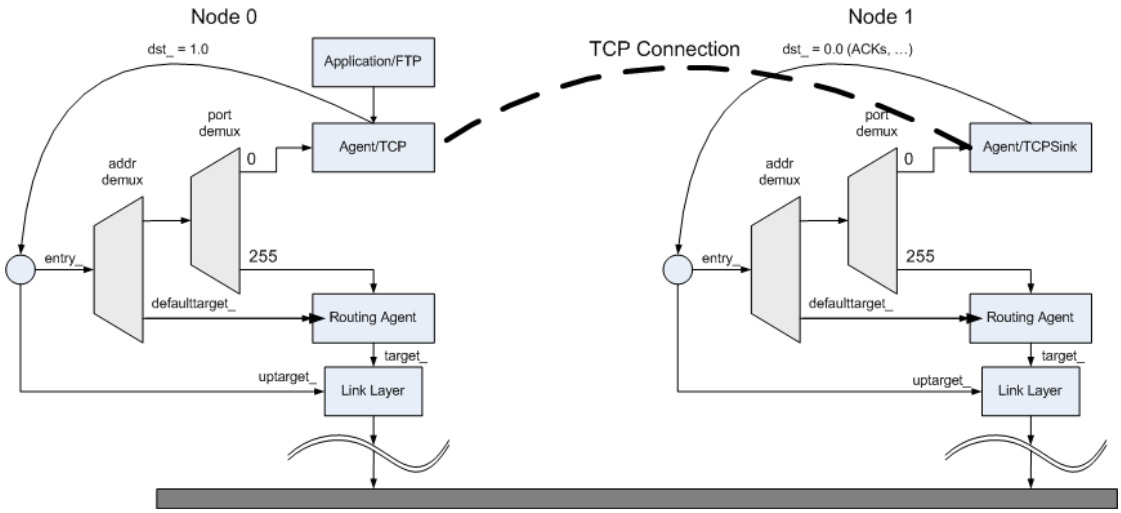


Figure 3.2: TCP connection between two wireless nodes in NS2

Listing 3.2: Trace file output for the script in Listing 3.1

```
1 s s -t 10.000000000 -Hs 0 -Hd -2 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -
    Ne -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 1.0 -
    It tcp -Il 40 -If 0 -Ii 0 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 0 -Po 0
2 r -t 10.004468822 -Hs 1 -Hd -2 -Ni 1 -Nx 60.00 -Ny 75.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 13a -Md 1 -Ms 0 -Mt 800 -Is 0.0 -Id
    1.0 -It tcp -Il 40 -If 0 -Ii 0 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 1 -
    Po 0
3 s -t 10.004468822 -Hs 1 -Hd -2 -Ni 1 -Nx 60.00 -Ny 75.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.0 -Id 0.0
    -It ack -Il 40 -If 0 -Ii 1 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 0 -Po 0

4 r -t 10.006305603 -Hs 0 -Hd -2 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 13a -Md 0 -Ms 1 -Mt 800 -Is 1.0 -Id
    0.0 -It ack -Il 40 -If 0 -Ii 1 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 1 -
    Po 0
5 s -t 10.006305603 -Hs 0 -Hd -2 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 1.0
    -It tcp -Il 1040 -If 0 -Ii 2 -Iv 32 -Pn tcp -Ps 1 -Pa 0 -Pf 0 -Po
    0
6 r -t 10.016242384 -Hs 1 -Hd -2 -Ni 1 -Nx 60.00 -Ny 75.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 13a -Md 1 -Ms 0 -Mt 800 -Is 0.0 -Id
    1.0 -It tcp -Il 1040 -If 0 -Ii 2 -Iv 32 -Pn tcp -Ps 1 -Pa 0 -Pf 1
    -Po 0
7 s -t 10.016242384 -Hs 1 -Hd -2 -Ni 1 -Nx 60.00 -Ny 75.00 -Nz 0.00 -Ne
```

```
    -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.0 -Id 0.0
    -It ack -Il 40 -If 0 -Ii 3 -Iv 32 -Pn tcp -Ps 1 -Pa 0 -Pf 0 -Po 0
8 r -t 10.018079165 -Hs 0 -Hd -2 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 13a -Md 0 -Ms 1 -Mt 800 -Is 1.0 -Id
    0.0 -It ack -Il 40 -If 0 -Ii 3 -Iv 32 -Pn tcp -Ps 1 -Pa 0 -Pf 1 -
    Po 0
```

3.2 Design Decisions

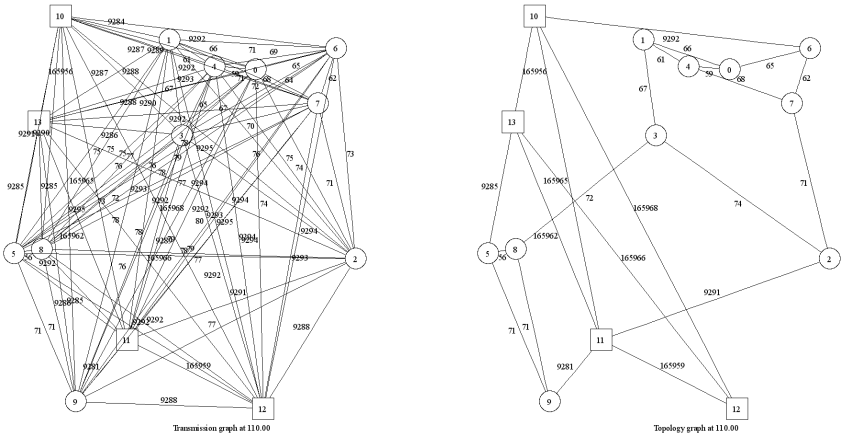
Implementing a new component opens up a great number of questions. To which layer does topology management belong? What is the impact on other layers? Should or do other layers need to know something about topology management? We came up with the following design decisions:

- The upper layers should have no knowledge about the TMA. In the author's opinion, this is a very reasonable restriction, because it provides a clear distinction between low- and high level services. And topology management clearly belongs to the lower network levels. As a consequence of this, routing algorithms, which run on top of this framework, are unaware of the real physical network layout.
- The decision if a packet should be dropped should be done as early as possible to reduce power usage[1] at the node. Therefore we have decided to perform packet level filtering at the MAC level, which is the first layer where network addresses are known and nodes can be identified.
- We decided on a very simply interface for the actual implementation of the topology management algorithm. The framework calls the algorithm implementation of either TMA::allow_up or TMA::allow_down, which in turn must return **true** if traffic from or to that neighbor is allowed. If **false** is returned, no traffic is allowed and the packet is dropped.

For example, consider the network shown in Figure 3.3. Each node is in the vicinity of each other, which can be seen by looking at the transmission graph in Figure 3.3(a). The network topology spawned by the TMA is shown by the edges which connect the nodes in the topology graph in Figure 3.3(b).

If a routing protocol would be executed on top of the topology graph, it would come to the conclusion that node 1's only neighbors are $0, 3$, and 4. Now assume that node 1 wants to exchange information with node 2. In the case of the TMA, a routing protocol would route the packet over node 3. If the routing protocol were executed on top of the transmission graph, almost every routing protocol would choose the direct connection. In a wired network, the direct connections would certainly make more sense (assuming

[1] Assuming that a node needs more power when computations are carried out. This is reasonable because modern power schemes require that the processor draws less power when no work is carried out.



(a) Transmission Graph (b) Topology Graph

Figure 3.3: Example network for $n = 10$ with Transmission- and Topology Graph

no special cases). But this situation is different in the wireless world. Using multihop communication can save power, because transmission power grows at least quadratic with the distance. Furthermore, the usage of the TMA can increase the network stability because only "good" connections are used. "Good" connections in this sense are connections which are long-living. It is also clear that if node movements are considered, connections which are of very poor quality will fail first. Changes in connections will typically trigger updates of routing protocols, at least if they are proactive. Therefore, the usage of a TMA is also beneficial for routing protocols.

3.3 Modifications made to NS2

NS2 does not provide any support for topology management. Therefore we had to rework some parts of the wireless model to allow our implementation to be as generic as possible. These modifications are shown in Figure 3.4. In our modified version, every packet is given to the topology management component called TMA. In NS2, the TMA component is defined by its implementation in the files `mac/tma.h` and `mac/tma.cc`. The TMA class itself is abstract, and an actual implementation must subclass it. Two examples are provided to simplify development of further TMA modules. The TMA /None, algorithm which does not affect the topology at all and which is defined in `mac/tma.cc`. A real implementation can be found in TMA/Filter , which is defined in `mac/tma-filter.cc` and declared in `mac/tma-filter.h`. The implementation of the concrete Thallner algorithm is quite complex and has been moved into a separate subdirectory `thallner`.

The link layer, TMA, IFQ, MAC and PHY components shown in Figure 3.4 are chained together in the TCL function add−interface in `tcl/lib/ns-mobilenode.tcl`.

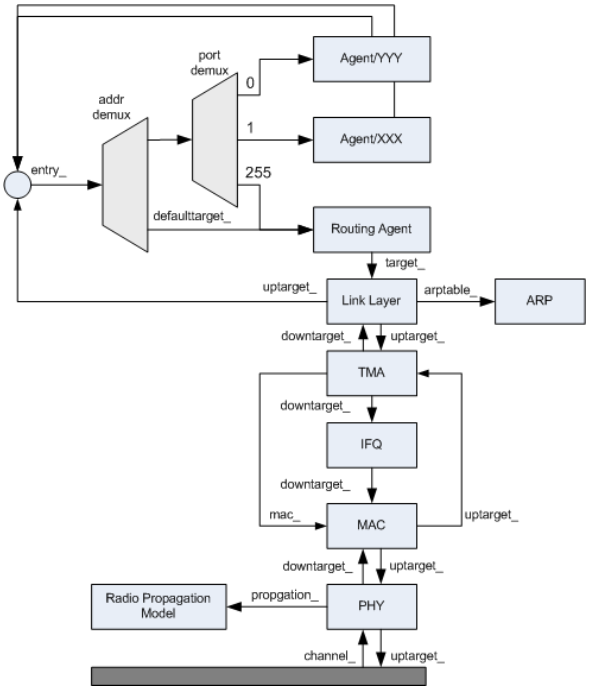


Figure 3.4: NS2 wireless model with TMA modifications

In addition , some modifications have been made in `tcl/ns-node.tcl`, `tcl/ns-lib.tcl`, and `tcl/lib/ns-default.tcl` to add the additional initialization functions. These files do not need to be changed, but the interested reader can take a look at the patches to see our modifications, although an advanced knowledge of the network simulator is required for this.

3.4 Implementation of the Thallner Algorithm

We have discussed the algorithmic implementation of the Thallner algorithm in Section 2.3. Going for a real world implementation is somewhat more difficult, because a lot of assumptions can not be met anymore. Furthermore, there are no communication primitives as *reliable multicast* readily available in NS2. We started by analyzing the network simulator and came up with the following list of missing components.

Reliable Transmission: Our network model assumes reliable transmission. NS2 uses a real world wireless network model and delivery of packets is not guaranteed. Therefore, we have implemented an ACK-Based transmission protocol to support reliable message delivery.

Multicast Communication: Almost all of our algorithms require multicast communication primitives. The IEEE 802.11 MAC layer upon which our TMA implementation works does not provide multicasting support. We therefore implemented two different multicast services, where the first one supports *Reliable Multicasting* using an ACK-based approach, and the second one is a very simple *Multicast Service* with very low overhead but without any quality of service features.

Neighbor Discovery: Our algorithms assume that the network neighborhood of a node is known. In our first version, we started with the implementation of two different neighborhood discovery protocols. During our studies we discovered that these modules were a strong performance limiting component, and we have moved this task into the link-state service.

Link State Service: A *Link State Service* supports the exchange of communication weights and links. It allows a node to retrieve information about its adjacent neighbors and about the state of the links of its neighbors. It is used by the propose modules to calculate new proposals.

NBAC: An *NBAC - Non Blocking Atomic Commitment* service which is used by the group checking and proposal algorithms. The NBAC supports our finalizing extension described in Section 2.3.6.

Serializing Component: Most of our algorithms use different types or records and require the usage of sets and vectors. These kinds of objects cannot be simply transmitted over the network. Therefore, these data structures must be serialized appropriately to wrap them into network messages. Problems here are the actual data encoding, the limited size of network messages, and endian issues. With the exception of the later, we have addressed all of them, and our implementation is flexible enough to allow further modifications.

We will discuss these components in greater detail in Section 3.6. We will start with a more practical approach and show how our components can be used and will present a complete working example. More examples are available in Appendix C. Note that the

next Section requires the reader to have finished our installation instructions shown in Appendix A. After our example, we will present the networking components, although we left out some implementation details. The final reference remains the source code and the doxygen documentation. Finally, we will continue in Section 3.7 with the discussion of the group checking and group construction modules.

3.5 Setup and Configuration

The topology management components support a number of configuration properties and commands. This section describes their usage, but does not focus on application examples. For examples, the reader is referred to Appendix C.

3.5.1 Enabling Topology Management in NS2

Topology management is enabled by changing the wireless node configuration in the NS2 setup scripts. A typical wireless node setup is shown in Listing 3.3. It is recommended to use a set of variables to protect against possible errors when the same information is used twice.

Listing 3.3: Enabling Topology Management in NS2

```
1 ...
2 set val(tma)              TMA/Filter
3 ...
4 $ns node-config \
5      ...
6      -tmaType $val(tma) \
7      -tmaTrace ON \
8      ...
```

-tmaType If this variable is not set or set to the empty string, no topology management is enabled. Otherwise, the variable should be set to one of the available TMA modules described below.

-tmaTrace If tracing at the TMA layer is required, one should set the value of −tmaTrace to ON. The default of −tmaTrace is OFF and no additional trace information is generated.

3.5.2 The TMA/Filter module

The TMA/Filter module serves as an example for developing new topology management modules for NS2. It needs a runtime configuration, because it drops all packets by default. An example is provided in Appendix C.4.3. The TMA/Filter module supports the following functions and settings.

Configuration settings

These variables should be set before any instances are created, that is, before $ns node is called. They are then used as default values for the class members.

debug_ If set to true, this creates additional debug output which is sent to stdout. To set it, simply add the line TMA/Filter **set** debug_ 1 somewhere before the first nodes are created.

.

Supported commands

add-neighbor args: This method takes a list of MAC addresses as argument, and every element in this list is added to the list of allowed nodes. For an example, consider the TCL code shown in Listing 3.4, which configures node 2 to allow packets from/to the nodes with MAC addresses $5, 8$, and 10[2].

Listing 3.4: Configure neighbors for TMA/Filter

```
1 # Get the TMA instance for node 2 and interface 0.
2 set tma [ $n(2) set tma_(0) ]
3 # $tma is now an instance of TMA/Filter
4 $tma add-neighbor 5 8 10
```

remove-neighbor args: This method takes a list of MAC addresses as argument and removes every neighbor in this list from the list of allowed neighbors.

clear-neighbors: Clears the list of allowed neighbors. After calling this function, no more packets will pass the topology management layer until new nodes are added again.

list-neighbors: Lists all allowed neighbors.

3.5.3 The TMA/None module

This module does not support any additional configuration. It permits all traffic and, depending upon the setting of TCL variable tmaTrace, it writes all traffic to the trace file.

3.5.4 The TMA/Thallner module

The TMA/Thallner module works out of the box using the default settings. After startup, the Thallner algorithm is executed and a new network topology is established.

Configuration settings

These variables should be set before any instances are created, that is, before $ns node is called. They are then used as default values for the class members. Boolean values should be set to 1 to turn them on or to 0 to turn them off.

debug_ If set to true, this creates additional debug output which is sent to stdout.

The following settings can only be changed by recompiling the source code. Since they do not need to be changed very often, this seemed like a reasonable approach.

EPSILON This values is the same as ϵ in the algorithms. It is defined in `tma-thallner.h` and should not be changed without a good reason. The default value is 0.1.

[2]Note that you can get the MAC address of a node by calling [$node **set** mac_(0)] id.

THALLNER_K The size of the members of a group. This corresponds to the value k used in the algorithms and is defined in `thallner/utils.h`. The default value is 3.

CHKGRP_NETWORK_LOAD_AVG The fraction of network resources allocated for group checking. This value is used by the group checking algorithm to schedule the checking of the groups. Setting this too high will result in network congestion[3]. It is defined in `thallner/tma-thallner.cc` and its default value is 0.05.

PROPOSE_NETWORK_LOAD_AVG The fraction of network resources allocated for group proposals. It uses the estimated time required for a proposal and the estimated number of nodes to calculate the time between the triggering of the propose module. Setting this too high will result in network congestion. It is defined in `thallner/tma-thallner.cc` and its default value is 0.1.

RMCAST_RETRY_TIMEOUT The time the reliable multicast service waits for an ACK message. Lower values give better performance but result in higher network load and more retries. If the value is set too high, the performance of algorithms using this service will suffer. The default value is 1 second. It is defined in `multicast-reliable.cc`.

RMCAST_MAX_RETRIES Maximum number of retries for a reliable multicast transmission. If a multicast initiator does not receive ACKs within the defined timeout it resends the request. If the number of retries is exceeded simulator execution is aborted and an assumption violation error is reported. The default value is 10. It is defined in `multicast-reliable.cc`.

RMCAST_RETRY_JITTER A jitter for sending ACK messages and for message retries. It is defined in `multicast-reliable.cc`. Whenever a node sends a multicast message the actual transmission is scheduled at now + Random::uniform()∗ RMCAST_RETRY_JITTER. This is specially important for the ACKs because they will be sent at the same time at all receiving nodes. The default value is 0.1 seconds.

SMCAST_SEND_JITTER A jitter for sending simple multicast messages. It is defined in `multicast-simple.cc`. The default value is 0 seconds.

EDGE_MIN_WEIGHT The minimum edge weight which is used during calculations. Set this to match your environment and your neighbor discovery protocol. The default value is 50.0, which is suitable for the default link-state service, and it should not be changed. It is defined in `tma-proposal.cc`

PROPOSAL_QUERY_TIMEOUT If the local non-perfect propose module needs link-state information from other nodes, it waits this time for the completion of the

[3] Too high in this context depends on the amount of other network load. We have not performed a lot of expirements with this value but < 0.1 is a safe value.

queries. Settings this too low will not allow the propose modules to find any proposals, because the search is aborted during runtime. The default value is 0.5 and it is defined in `tma-lnp-proposal.cc`.

LSB_FULLUPDATE_INTERVAL The time between full update messages sent by the link-state service. These update messages are used to inform other nodes about the state of the links. Setting this too low results in a high and constant network load. Setting this to high will reduce performance in case of node movements. The default value is 5 seconds, and it is defined in `linkstate-basic.cc`.

LSB_HELLO_INTERVAL The time between hello messages. Hello messages are used by a node to detect that a neighbor is alive. During this time, only partial updates are sent, which are shorter in size than full update message. The default value is 1 second, and it is defined in `linkstate-basic.cc`.

PROPOSAL_ATTACH_ID Attach an ID to every proposal. This makes it easier to debug the propose module, but increases message size. It is a boolean and defined in `tma-proposal.h`.

PROPOSAL_ATTACH_CREATION_TIME Attach the time when a search was initiated. This can be used to obtain information on the typical time required for a proposal. If enabled, this information is available from the statistics interface. It is a boolean and defined in `tma-proposal.h`.

The following settings can be used to control debug information. A list of log levels is given in Table 3.1. The values should be or-ed together to enable multiple levels using the operator | in C++.

THALLNER_PROPOSAL_TRACE Amount of debug information returned by the group construction algorithm. Enable this if you want to debug or analyze the construction of the groups. Note that this debug information only affects already released proposals, which are passed to the main algorithm for construction. The default debug level is TRACE_LEVEL_ERROR and is defined in `thallner/tma-thallner.cc`.

THALLNER_CHECKGROUP_TRACE Amount of debug information returned by the group checking algorithm. Enable this if you want to debug or analyze group checking. The default value is TRACE_LEVEL_ERROR and is defined in `thallner/tma-thallner.cc`.

RMCAST_TRACE Can be set to configure the level of information reported by the reliable multicasting module. The default is TRACE_LEVEL_ERROR which only reports critical errors. Further debug information can be enabled by adding additional log levels. It is defined in `thallner/multicast-reliable.cc`.

SMCAST_TRACE Can be set to configure the level of information reported by the simple multicasting module. The default is TRACE_LEVEL_ERROR which only

reports critical errors. Further debug information can be enabled by adding additional log levels. It is defined in `thallner/multicast-simple.cc`.

LSB_TRACE Controls whether information about link-state queries should be logged. The default is TRACE_LEVEL_ERROR which does not generate any information. It is defined in `thallner/linkstate-basic.cc`.

LSB_MSG_TRACE Controls whether information about messages sent and received should be logged. The default is TRACE_LEVEL_ERROR which does not generate any information. It is defined in `thallner/linkstate-basic.cc`.

LSB_UPDATE_TRACE Controls whether information about new neighbors should be logged. Anytime a neighbor information is updated some trace information is produced. The default is TRACE_LEVEL_ERROR which does not generate any information. It is defined in `thallner/linkstate-basic.cc`.

NBAC_TRACE Controls whether the non-blocking atomic commitment service should generate debug output. The default value is TRACE_LEVEL_ERROR, which only reports critical errors. Further debug information can be enabled by adding additional log levels. It is defined in `nbac.cc`.

PROPOSAL_TRACE Controls whether the execution of the propose module should generate debug information. The default value is TRACE_LEVEL_ERROR, which only reports critical errors. It is defined in `tma-proposal.h`.

PROPOSAL_CALCULATE_TRACE Controls whether the calculations performed by the propose module should be traced. Only enable this if you think you have found a bug in the implementation of the basic functions, or for studying purposes. The default value is TRACE_LEVEL_ERRORS, and it is defined in `tma-proposal.h`.

PROPOSAL_PMS_TRACE Controls whether the calculation of the potential member set should be traced. Enable this if you think that a group can be constructed but it is not. The default value is TRACE_LEVEL_ERRORS, and it is defined in `tma-proposal.h`.

Table 3.1: Configurable log levels for modules

Log level	*Description*
TRACE_LEVEL_DEBUG_L0	Debug information. Disable for simulations.
TRACE_LEVEL_DEBUG_L1	Additional debug information. Disable for simulations.
TRACE_LEVEL_INFO_L0	Informational messages. Disable for simulations.
TRACE_LEVEL_INFO_L1	Informational messages. Disable for simulations.
TRACE_LEVEL_WARNING	Warning messages. Disable to improve performance.
TRACE_LEVEL_ERROR	Error messages. Do not disable.
TRACE_LEVEL_ALL	Enable all loglevels. Use with care.

Supported commands

tmaaddr Returns the TMA address of this node.

mac Returns the TCL object for the MAC layer. If called with an argument it allows the setting of the MAC layer. Setting the MAC layer is considered an internal function and it should not be used by the user.

phy If called with an argument, it allows the setting of the physical layer. This method is used internally and should not be used.

log-target Attach a trace object to the TMA. This method is used internally and should not be used.

tracetarget Attach a trace object to the TMA. This method is used internally and should not be used.

start Starts execution of the TMA at this node. This method should be executed at every node immediately after startup.

node-degree Returns the node degree, i.e., the number of active connections created by the TMA.

last-change Returns the time in seconds when the topology has last changed at this node. This can be used to implement a convergence detection for the Thallner algorithm.

proposals-pending Returns the number of currently pending proposals known to this node.

proposals-period Average time between the generation of new group proposals.

groupcheck-period Average time between the checking of groups at a node.

gateway-node BOOLEAN If called without an argument, it returns 1 if this node is a gateway node, otherwise 0. If called with an argument, the type of this node can be changed. Changing the type of the node is only supported before execution has been started.

thallner-k The value of k used within this run.

tmaaddr Returns the TMA address of this node.

stats Prints statistical information about this node. This includes information from the multicast components, from the link-state service, and from the NBAC component.

stats MODULE PROPERTY ARGS Query the module MODULE for statistical information about the property PROPERTY. A list of all properties together with their description are shown in Table 3.2. This can be used to compute average or total information from various nodes.

dump-topology-tree FILENAME Uses global information to generate the complete topology tree and writes it to the file `FILENAME`. It should be processed by the `dot` tool from the GraphViz suite. If this function is called and groups are currently under construction, the information might not be consistent, because the global information is obtained by merging all local information.

dump-local-topology-tree FILENAME Dumps the local topology tree at this node into the filename `FILENAME`. The output file written should be processed by the `dot` tool from the GraphViz suite.

dump-transmission-graph FILENAME Generates a transmission graph which should be processed by the `neato` tool from the GraphViz suite.

dump-neighbor-graph FILENAME Generates a graph from the node and all its adjacent neighbors. This graph includes the edge weights. It should be processed by the `neato` tool from the GraphViz suite.

dump-topology-graph FILENAME Uses global information to dump the complete topology graph and writes it to the file `FILENAME`. The output should be processed by the `neato` tool from the GraphViz suite.

dump-local-topology-graph FILENAME Dumps the node local topology graph. In case the topology is converged or it is not subject to change this graph is a subgraph of the topology graph. The output should be processed by the `neato` tool from the GraphViz suite.

dump-spanner FILENAME Generates a spanning tree for the transmission graph. It should be processed by the `neato` tool from the GraphViz suite.

Table 3.2: Statistical information from modules

Module	*Property*	*Type*	*Args*	*Description*
	msg_sent	int	port(1,int)	Number of messages sent using application port '`port`'.
	msg_sent_dup	int	port(1,int)	Number of messages resent because of missing ACKs.
rmcast	msg_received	int	port(1,int)	Number of messages received for application port '`port`'.
	msg_received_dup	int	port(1,int)	Number of message duplicates received for application port '`port`'.
	acks_sent	int	port(1,int)	Number of ACKs sent.
	acks_received	int	port(1,int)	Number of ACKs received.
smcast	msg_sent	int	port(1,int)	Number of messages sent at using application port '`port`'.
	msg_received	int	port(1,int)	Number of messages received for application port '`port`'.
	initiated	int	msg_type(1,int)	Number of NBAC instances of type '`msg_type`' initiated at this node .
	participated	int	msg_type(1,int)	Number of NBAC instances of type '`msg_type`' this node has participated in.
nbac	commited	int	msg_type(1,int)	Number of commited NBAC instances of type '`msg_type`' initiated by this node.
	aborted	int	msg_type(1,int)	Number of aborted NBAC instances of type '`msg_type`' by this node.
	out_of_order	int	msg_type(1,int)	Number of NBAC messages which where received out of order.
	checks_done	int		Number of group checks initiated at this node.
	diameter	int		The diameter of the topology graph. Use this information only after the network has converged.
	proposals_done	int		Number of proposals done at this node.
	packets_other	int		Number of normal network packets.
	packets_tma	int		Number of packets related to TMA.
tma	packets_dropped_up	int		Number of packets dropped by the TMA which should be passed up to the link-layer.
	packets_passed_up	int		Number of packets passed to the link-layer.
	packets_dropped_down	int		Number of packets dropped by the TMA which should have been passed down to the MAC layer.
	packets_passed_down	int		Number of packets passed to the MAC layer.
	average_proposal_time	double		Average time required for a proposal.
propose	initiated	int		Number of proposals initiated at this node.
	released	int		Number of proposals released to the TMA at this node.
lsb	full_updates_sent	int		Number of full update messages sent by the basic link-state service.
lsb	partial_updates_sent	int		Number of partial updates sent by the basic link-state service.
lsb	full_updates_received	int		Number of full update messages received.
lsb	partial_updates_received	int		Number of partial update messages received.
lsb	pwr_saving_tma	list		Returns the minimum, maximum, and average transmission power in dBm required for all neighbors in the topology induced by the TMA. It is returned as a TCL list min max avg.
lsb	pwr_saving_other	list		Returns the minimum, maximum and average transmission power in dBm required for all neighbors. It is returned as a TCL list min max avg.

3.5.5 Example

This example shows the most basic setup using the TMA algorithm. It contains 10 normal nodes and 3 gateway nodes , and the final results are shown in Figures 3.6 and 3.5. The script used to setup the simulator is shown in Listing 3.5. The first part of the script shown in lines 17−84 sets simulation parameters and configures the simulator. The second part shown in lines 86 − 183 contains utility functions to detect the convergence of the algorithm and to generate the output files. In the third part, the wireless nodes are created in lines 185 − 215. Step four initializes the required global variables used by the utility functions, configures three nodes as gateway nodes, and starts the TMA at time 0.0 at every node. This happens in lines 217 − 231. The last part sets up the scheduler and starts the simulation. After execution, the simulation will have generated the output files `topology-tree-t81-dot.dot`, `topology-graph-t81-neato.dot`, and `adhoc.tr`, which can be analyzed by the appropriate tools. The simulation can be started by executing 'make trace', and png and eps images can be generated by calling 'make images'.

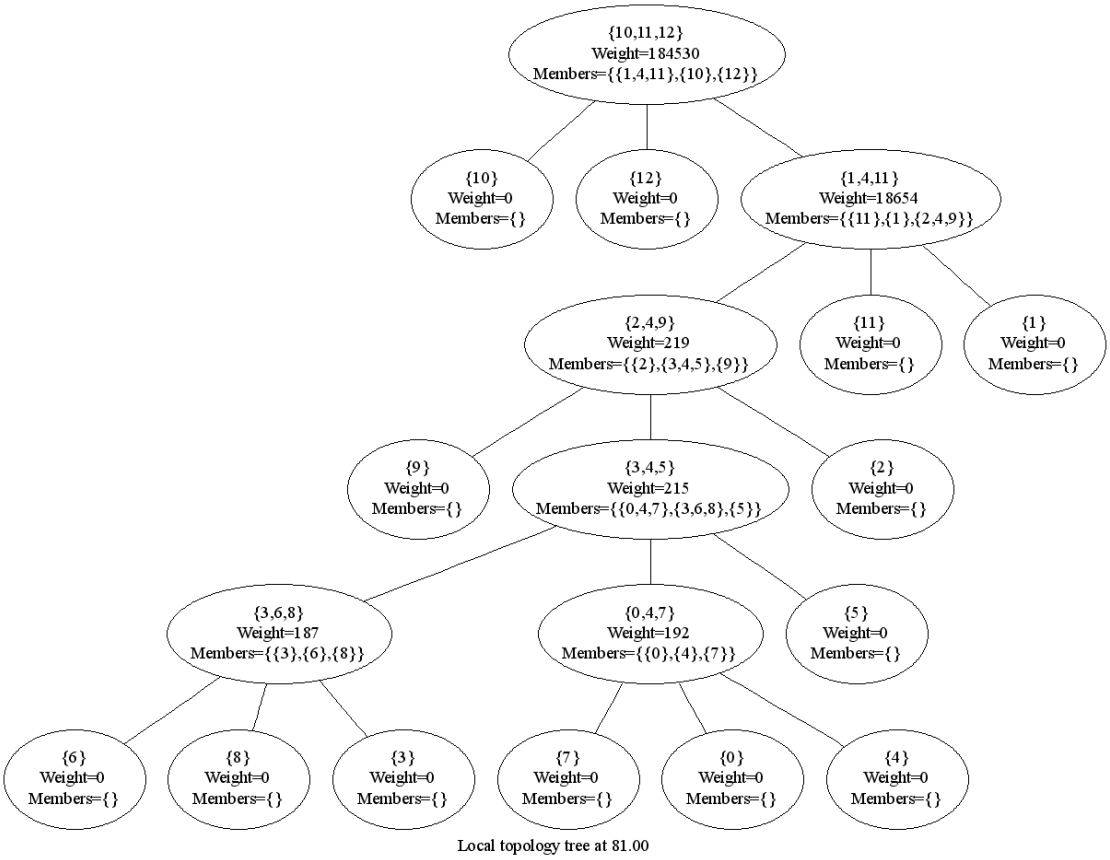


Figure 3.5: Topology tree after execution for $n' = 10$ and $n'' = 3$

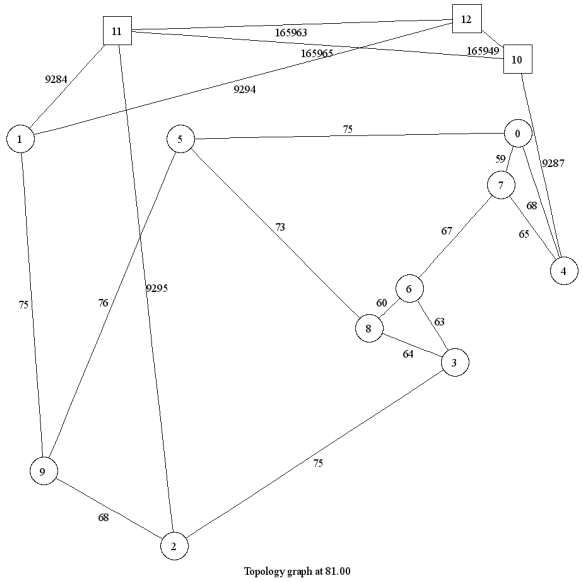


Figure 3.6: Network topology graph for $n' = 10$ and $n'' = 3$

Listing 3.5: NS2 simulator setup script

```
1  # ----------------------------------------------------------------------
2  # Project: Template for NS2 simulation framework
3  # Author:  Christian Walter <e0225458@student.tuwien.ac.at>
4  #
5  # $Log: ns2-setup-adhoc.tcl,v $
6  # Revision 1.3  2007/10/10 19:49:00  cwalter
7  #  - Final updates.
8  #
9  # Revision 1.2  2007/07/29 21:36:33  cwalter
10 #  - New code examples for ns2 chapter.
11 #
12 # Revision 1.1  2007/07/29 11:24:05  cwalter
13 #  - Basic TMA example.
14 #
15 # ----------------------------------------------------------------------
16
17
18 # ----------------------------------------------------------------------
19 # set configuration properties
20 # ----------------------------------------------------------------------
21 set num_nodes               20                          ;# total nodes
22 set xsize                   100
23 set ysize                   100
24
25 set val(chan)               Channel/WirelessChannel     ;# Channel Type
```

```
26 set val(prop)            Propagation/TwoRayGround    ;# radio-propagation
       model
27 set val(netif)           Phy/WirelessPhy             ;# network interface
       type
28 set val(mac)             Mac/802_11
29 set val(tma)             TMA/Thallner
30 set val(ifq)             Queue/DropTail/PriQueue     ;# interface queue type
31 set val(ll)              LL                          ;# link layer type
32 set val(ant)             Antenna/OmniAntenna         ;# antenna model
33 set val(ifqlen)          100                         ;# max packet in ifq
34 set val(rp)              DumbAgent
35 set last_stats_time      0.0
36
37 #===================================================================
38 # Configure for IEEE802.11b
39 #===================================================================
40 Mac/802_11 set SlotTime_           0.000020          ;# 20us
41 Mac/802_11 set SIFS_               0.000010          ;# 10us
42 Mac/802_11 set PreambleLength_     144               ;# 144 bit
43 Mac/802_11 set PLCPHeaderLength_  48                ;# 48 bits
44 Mac/802_11 set PLCPDataRate_       1.0e6             ;# 1Mbps
45 Mac/802_11 set dataRate_           11.0e6            ;# 11Mbps
46 Mac/802_11 set basicRate_          1.0e6             ;# 1Mbps
47
48 Phy/WirelessPhy set freq_          2.4e+9            ;# Frequency
49 Phy/WirelessPhy set Pt_            3.3962527e-2      ;# Transmission power
50 Phy/WirelessPhy set RXThresh_      6.309573e-12      ;# Receiver threshold
51 Phy/WirelessPhy set CSThresh_      6.309573e-12      ;# Sense threshold
52
53 #===================================================================
54 # Create a simulator instance
55 #===================================================================
56 set ns [ new Simulator ]
57 set tracefd [ open adhoc.tr w ]
58 $ns use-newtrace
59 $ns trace-all $tracefd
60
61 set topo [ new Topography ]
62 $topo load_flatgrid $xsize $ysize
63
64 set god_ [ create-god $num_nodes ]
65
66 set chan_1_ [new $val(chan)]
67 $ns node-config \
68      -adhocRouting $val(rp) \
69      -llType $val(ll) \
70      -macType $val(mac) \
71      -tmaType $val(tma) \
72      -phyType $val(netif) \
73      -ifqType $val(ifq) \
74      -ifqLen $val(ifqlen) \
75      -antType $val(ant) \
76      -propType $val(prop) \
77      -topoInstance $topo \
```

```
78      -agentTrace ON \
79      -routerTrace OFF \
80      -macTrace OFF  \
81      -tmaTrace OFF \
82      -movementTrace OFF \
83      -channel $chan_1_
84
85
86
87  #-----------------------------------------------------------------
88  # Common functions
89  #-----------------------------------------------------------------
90  proc finish {} {
91      global ns tracefd
92      $ns flush-trace
93      close $tracefd
94      exit 0
95  }
96
97  proc convergence-calculate-times { arrname } {
98    global tma topo_thallner_k
99    upvar $arrname times
100
101   set num_nodes [ array size tma ]
102   set topo_thallner_k [ $tma(0) thallner-k ]
103
104   # floor( (n-1)/(k-1) ) = ngroups is the number of groups. If the
105   # tree is balanced the height is log( ngroups ) / log( k ). Every
106   # node therefore walks up this hierarchy and creates group proposals
107   # where 1 is added because of the proposal for the node itself.
108   set group_tree_depth [ expr log( ( $num_nodes - 1 )/( $topo_thallner_k -
          1 ) ) / log( $topo_thallner_k ) + 1 ]
109
110   # add an extra factor of 2 because a generated proposal must also
111   # be accepted by the thallner algorithm.
112   set times(convergence_time_proposals) [ expr [ $tma(0) proposals-period ]
           * $group_tree_depth * 2.0 ]
113
114   # calculate the time required to check all groups at a node
115   set times(convergence_time_group) [ expr [ $tma(0) groupcheck-period ] *
          $group_tree_depth ]
116
117   # the convergence time is the maximum
118   if { $times(convergence_time_group) < $times(convergence_time_proposals)
          } {
119     set times(convergence_time) $times(convergence_time_proposals)
120   } else {
121     set times(convergence_time) $times(convergence_time_group)
122   }
123 }
124
125 proc convergence-test {} {
126   global ns tma val last_stats_time topo_prefix
127   set scheduler [ $ns set scheduler_ ]
```

```
128   set now [ $scheduler now ]
129   set topo_thallner_k [ $tma(0) thallner-k ]
130
131   convergence-calculate-times times
132
133   # we assume that the network has converged.
134   set is_stable 1
135
136   # compute the time when the group structures have changed.
137   set last_change_max 0.0
138   for { set i 0 } { $i < [ array size tma ] } { incr i } {
139     set last_change [ $tma($i) last-change ]
140     if { $last_change > $last_change_max } {
141       set last_change_max $last_change
142     }
143     # if this node is not a gateway node it must have a node
144     # degree of k.
145     if { 0 == [ $tma($i) gateway-node ] } {
146       if { $topo_thallner_k != [ $tma($i) node-degree ] } {
147         set is_stable 0
148       }
149     }
150   }
151
152   # if the group structures have not changed during the convergence
153   # time the network is stable.
154   if { $now > $times(convergence_time) } {
155     if { $now - $last_change_max < $times(convergence_time) } {
156       puts "group_structure_has_changed_at_time_$last_change"
157       set is_stable 0
158     }
159   } else {
160     set is_stable 0
161   }
162
163   if { $is_stable == 1 } {
164     dump-topology-graph
165     dump-topology-tree
166     $ns at [ expr $now + 0.01 ] "finish"
167   } else {
168     $ns at [ expr $now + 1 ] "convergence-test"
169   }
170 }
171
172 proc dump-topology-graph {} {
173   global tma ns
174   set now [ [ $ns set scheduler_ ] now ]
175   $tma(0) dump-topology-graph topology-graph-t[ expr round($now) ]
        -neato.dot
176 }
177
178 proc dump-topology-tree {} {
179   global tma ns
180   set now [ [ $ns set scheduler_ ] now ]
```

```
181   $tma(0) dump-topology-tree topology-tree-t[ expr round($now) ]-dot.dot
182 }
183
184
185
186 #---------------------------------------------------------------------------
187 # Create network topology
188 #---------------------------------------------------------------------------
189 set node_(0) [ $ns node ]
190 $node_(0) set X_ 90; $node_(0) set Y_ 76;
191 set node_(1) [ $ns node ]
192 $node_(1) set X_ 3; $node_(1) set Y_ 75;
193 set node_(2) [ $ns node ]
194 $node_(2) set X_ 30; $node_(2) set Y_ 4;
195 set node_(3) [ $ns node ]
196 $node_(3) set X_ 79; $node_(3) set Y_ 36;
197 set node_(4) [ $ns node ]
198 $node_(4) set X_ 98; $node_(4) set Y_ 52;
199 set node_(5) [ $ns node ]
200 $node_(5) set X_ 31; $node_(5) set Y_ 75;
201 set node_(6) [ $ns node ]
202 $node_(6) set X_ 71; $node_(6) set Y_ 49;
203 set node_(7) [ $ns node ]
204 $node_(7) set X_ 87; $node_(7) set Y_ 67;
205 set node_(8) [ $ns node ]
206 $node_(8) set X_ 64; $node_(8) set Y_ 42;
207 set node_(9) [ $ns node ]
208 $node_(9) set X_ 7; $node_(9) set Y_ 17;
209 set node_(10) [ $ns node ]
210 $node_(10) set X_ 90; $node_(10) set Y_ 89;
211 set node_(11) [ $ns node ]
212 $node_(11) set X_ 20; $node_(11) set Y_ 94;
213 set node_(12) [ $ns node ]
214 $node_(12) set X_ 81; $node_(12) set Y_ 96;
215
216
217
218 #---------------------------------------------------------------------------
219 # Get an instance to the TMA objects and obtain the MAC address of each
220 # node.
221 #---------------------------------------------------------------------------
222 for { set i 0 } { $i < [ array size node_ ] } { incr i } {
223   set tma($i) [ $node_($i) set tma_(0) ]
224   set mac($i) [ [ $node_($i) set mac_(0) ] id ]
225   $ns at 0.0  "$tma($i) start"
226 }
227
228 $tma(10) gateway-node 1
229 $tma(11) gateway-node 1
230 $tma(12) gateway-node 1
231
232
233 #---------------------------------------------------------------------------
234 # Create traffic patterns
```

```
235 # ------------------------------------------------------------------------------
236
237 # ------------------------------------------------------------------------------
238 # Time schedule
239 # ------------------------------------------------------------------------------
240 $ns at 5.0 "convergence-test"
241 $ns run
```

3.6 Basic NS2 networking components

3.6.1 Multicast Service

We defined a common set of operations which must be supported by every multicasting component. These basic services include the possibilities for sending and receiving multicast messages and the retrieval of statistical information. The interface functions are shown in Figure 3.7 in the class 'Multicast'. Messages can be sent by calling the method 'multicast' with a set of destination addresses, an application specific port and the message payload encapsulated in the abstract data type 'data_t'. Every multicast message is assigned a unique multicast message identifier, briefly UID, which is represented by the abstract base class 'mcast_uid'. The class 'MulticastTarget' should be implemented by a component which needs to receive multicast messages. It allows a more generic implementation, because all receivers of multicast messages can be treated the same if they implement a common interface.

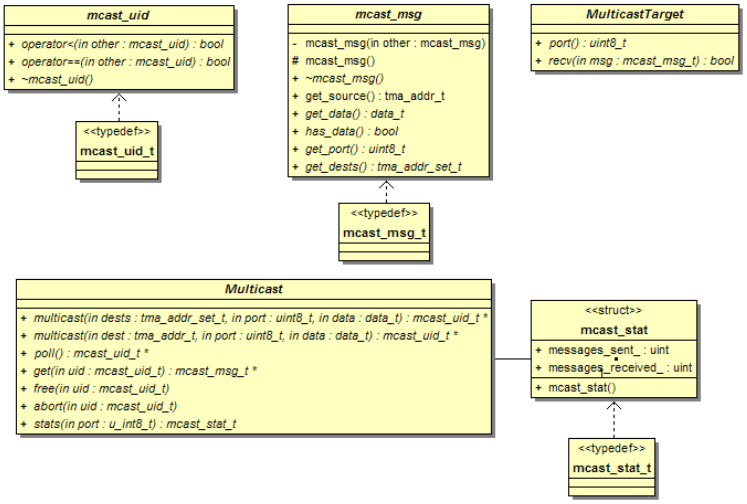


Figure 3.7: UML class diagram for multicast service

We will now show two basic examples on how to send messages and how information can be retrieved from the multicast service. In both examples, we assume that there exists an instance 'mcast_' of type 'Multicast'. A sender simply constructs a new message with an appropriate payload and passes it to the multicast service. This is shown in Listing 3.6. The code is self explanatory.

Listing 3.6: Sending of a multicast message

```
1 void BasicMulticastSendTest( )
2 {
3   ...
4   // create an output stream and serialize a simple 8 bit integer.
5   ostringstream ostr( ios::out | ios::binary );
```

```
6     msg_data_u_int8( 10 ).serialize( ostr );
7
8     // make a data payload object from it
9     std::string payload_str = ostr.str( );
10    data_t payload( reinterpret_cast< const u_int8_t *>( payload_str.c_str( )
          ), payload_str.length( ) );
11
12    tma_addr_set_t participants;
13    participants.insert( tma_addr_t( 5 ) );
14    participants.insert( tma_addr_t( 3 ) );
15    mcast_uid_t *uid = mcast_->multicast( participants, 0, payload );
16    ...
17  }
```

Retrieving the messages is a bit more difficult. First of all, we have chosen to implement a polling approach at the receiver. That is, messages are queued internally in the multicasting component and the receiver can take them out from the queue whenever it wants. This approach has a lot of benefits against a callback based approach. First of all, in a callback based approach the control flow is by the sender, and if messages must be received in a certain order, the function which would be called would have to perform the queuing by itself. This would result in duplicate code and possibly more bugs. Another benefit is that timeouts can be implemented quite easily, because a receiver would simply poll its multicast service, and if no messages have been received, it can simply treat this as an error. A basic usage of the multicasting service is shown in Listing 3.7. This simply checks if a message is in the queue and takes it out of the queue, handles it, and finally frees its resources.

Listing 3.7: Receiving a multicast message

```
1  void BasicMulticastReceiveTest( )
2  {
3    Multicast *mcast_ = NULL;
4
5    for ( ;; )
6    {
7      // Poll the stack for any events which need our attention.
8      std::auto_ptr< mcast_uid_t > uid( mcast_->poll( ) );
9      if ( uid.get( ) != NULL )
10     {
11       // retrieve the message from the multicast stack.
12       std::auto_ptr< mcast_msg_t > msg( mcast_->get( *uid ) );
13
14       // handle message and do something with the payload
15       data_t payload = msg->get_data( );
16
17       // free the resources
18       mcast_->free( *uid );
19     }
20     else
21     {
22       // no more messages
23       break;
```

```
24        }
25    }
26 }
```

3.6.2 Reliable Multicasting

Levine and Aceves made an excellent presentation of current reliable multicast protocols available in [LGLA98]. Following their work and using their excellent introduction, we came up with the following requirements: A must for every multicast service is that packets from a source are delivered to the receivers within a finite amount of time and free of errors. It is typically also required that packets can be deleted safely at the source within a finite amount of time, because memory is a limited resource. Additional requirements are that packets are delivered only once and in order.

There are two popular approaches, called *sender-initiated* and *received-initiated* [LGLA98, p.1]. ACK based protocols belong to the sender-initiated protocols, where it is in the responsibility of the sender to maintain the state of the receivers. This includes the set of destinations and the already received ACKs. For every packet received, the receiver unicasts an ACK back to the sender. This is shown in Figure 3.8(a). If all ACKs have been received, the sender can release the resources needed for keeping the state and knows for sure that every receiver has received the message.

NAK based protocols are called receiver-initiated protocols. It is the responsibility of the receiver to detect missing packets, for example by a gap in the sequence numbers, and to send a NAK back to the sender to force a retransmit. An example is shown in Figure 3.8(b), where the messages with ids 3 and 4 are retransmitted.

Both of these protocols have drawbacks. One of the known problems of sender-initiated protocols is the so called ACK implosion problem. This problem is a result of the fact, that every node needs to acknowledge a message. Therefore the number of ACKs increases with the number of multicast participants. A further problem are unreliable links in which case a lot of retransmits and ACKs are needed. For more information on this the reader is referred to [LGLA98]. Because of our special environment we have chosen to use an ACK based approach because the number of participants is limited by k^2 in our case which is reasonably small.

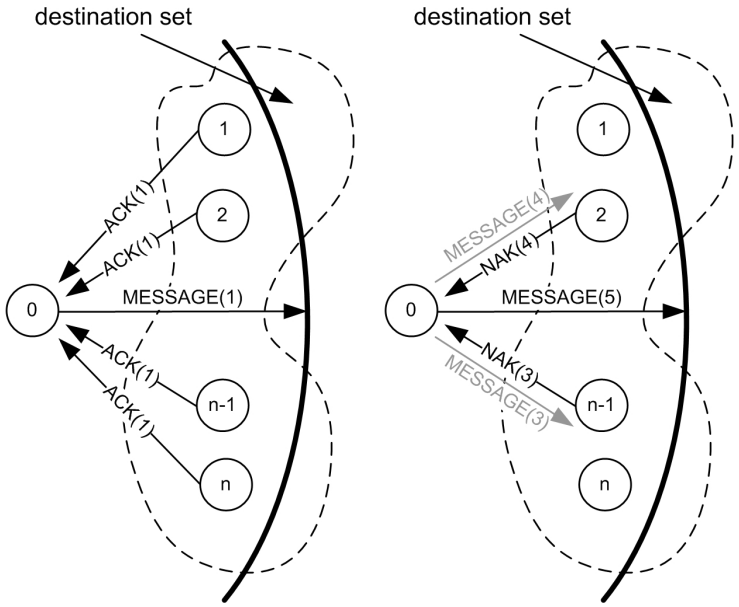
Definition 29 (Reliable Multicast)**.** *Our implementation of our* Reliable Multicast *protocol follows an ACK based approach and guarantees the following properties:*

Reliable: *A message sent by a sender is eventually delivered.*

No resource leaks: *A message sent by a sender is eventually freed at the sender if the destinations nodes are correct.*

Finite delivery: *If the destination nodes are in the transmission range of the sender, then a message sent by the source is eventually delivered at the destination within a finite amount of time.*

FIFO Multicast: *Messages sent by a sender are received in order by the receiver.*



(a) ACK based protocol (b) NAK based protocol

Figure 3.8: Sender- and Receiver initiated multicast protocols

Every message is stamped with a unique identifier, briefly called message ID, used to distinguish between multiple message. These message IDs together with relational operators are implemented in ' identifier_uid ' which subclasses 'mcast_uid'. The UML class diagram together with the most important operations is shown in Figure 3.9. Two important concepts are shown there: First we see the extension of 'mcast_uid' using the TMA addresses and the sequence counters to build unique IDs. The second addition are two more methods. The method 'recv' is used to dispatch NS2 packets into the multicast service. In case of a receiver, the multicast service will store the message in its internal queue such that it can be retrieved by a client and will generate an ACK message for the sender. At the sender site, the packet header is analyzed, and if such a multicast message is in transmit, the ACK is added to the list of received ACKs. If all ACKs have been received, the resources are released. In case of missing ACKs, the sender must periodically retransmit messages. This is implemented in the method 'pool' which is called by a periodic timer.

Frame Formats

The implementation of the reliable multicast works directly above the MAC layer, and no additional overhead is generated. The basic structure of an IEEE802.11 frame is shown in Table 3.3. A very good introduction to this subject is given in [Sch00, p.207-p.239]. It can be seen that there is no place to support multicasting at MAC level, at least not if

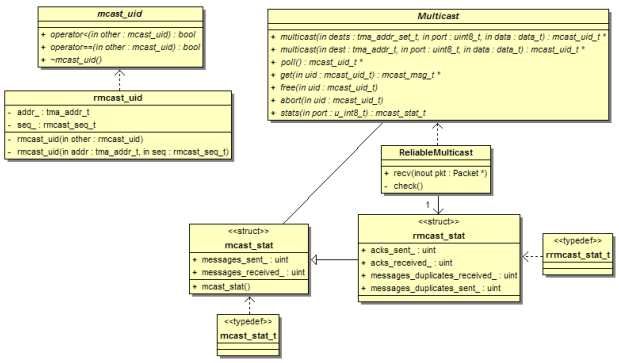


Figure 3.9: UML class diagram for reliable multicast service

no special multicast addresses are introduced. We therefore have decided to include the

Table 3.3: IEEE 802.11 MAC Data Frame Format

FC	*Duration/ID*	*Addr1*	*Addr2*	*Addr3*	*SC*	*Addr4*	*Frame Body*	*CRC*

FC . . . Frame Control.
SC . . . Sequence Control.
Addr1 . . . Address-1 is always the recipient address, i.e the immediate recipient of the packet. We will write DA for destination address which is our basic case.
Addr2 . . . Address-2 is always the transmitter address, i.e. the station physically transmitting the packet. We will write SA for source address which is our basic case.

set of recipients within the data payload, and all packets are sent to the MAC broadcast address. A receiving station must then perform a fast compare on all of the address fields (up to k^2) to decide if the packet should be dropped or if it is for this node. We map MAC addresses to TMA addresses by simply taking the decimal equivalent of them. Therefore `43:12:78:43:89:03` becomes 73746606164227. We distinguish between two different types of packets. The first one are REQUEST packets which are shown in Table 3.4.

Table 3.4: Reliable Multicast REQUEST packet

MAC header				Data							CRC
. . .	DA	SA	. . .	Type	Sequence	Port	Dest[0]	. . .	Dest[k-1]	Payload	. . .
. . .	DA	SA	. . .	0	$0 - 2^{32}$	$0 - 255$	Dest[0]	. . .	Dest[k-1]	Payload	. . .

SA . . . `OR:OR:OR:OR:OR:OR`.
DA . . . `FF:FF:FF:FF:FF:FF`.
Type . . . `RMCAST_PKT_TYPE_REQUEST = 0`.
Sequence . . . Sequence number at the sending node as an unsigned 32 bit integer.
Port . . . Application specific port used for message multiplexing.
Dest[i] . . . Up to k destinations where every field contains a MAC address. If less than k recipients are addressed unused fields are set to `FF:FF:FF:FF:FF:FF`.
Payload . . . The application payload, i.e. the actual data received by the nodes.

The second type of packets are ACK packets and are shown in Table 3.5. Contrary to REQUEST packets, ACK packets are sent as unicast packets, and do not include any data.

Table 3.5: Reliable Multicast ACK packet

MAC header				Data			CRC
...	DA	SA	...	Type	Sequence	Port	...
...	DS:DS:DS:DS:DS:DS	OR:OR:OR:OR:OR:OR	...	1	$0 - 2^{32}$	$0 - 255$	...

DA The MAC address of the node which should receive the ACK.
Type ... `RMCAST_PKT_TYPE_ACK = 1`.
Sequence ... Sequence number at the sending node.

3.6.3 Simple Multicasting

The simple multicasting service can be used when no reliable transmission is required, that is, when there is some sort of error detection or recovery available in the upper layers of the protocol. The UML class diagram together with the most important operations are shown in Figure 3.10. Two important concepts are shown there: First we see the extension of 'mcast_uid'. Contrary to the reliable multicasting service, the IDs are used only locally to identify messages. In addition, we can see the method recv which is used to dispatch NS2 packets into the multicast service.

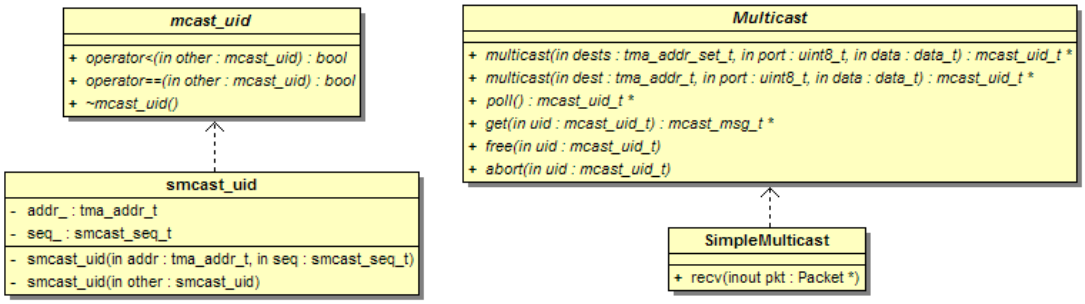


Figure 3.10: UML class diagram for simple multicast service

Frame Formats

The simple multicasting service only uses a single type of packet because no ACKs or other control messages are used. Its structure is shown in Table 3.6.

Table 3.6: Simple Multicast REQUEST packet

MAC header				Data					CRC
...	DA	SA	...	Port	Dest[0]	...	Dest[k-1]	Payload	...
...	FF:FF:FF:FF:FF:FF	SA	...	$0 - 255$	Dest[0]	...	Dest[k-1]	Payload	...

SA ... `OR:OR:OR:OR:OR:OR`.
Port ... Application specific port used for message multiplexing.
Dest[i] ... Up to k destinations where every field contains a MAC address. If less than k recipients are addressed unused fields are set to `FF:FF:FF:FF:FF:FF`.
Payload ... The application payload, i.e. the actual data received by the nodes.

3.6.4 Link-State Service

The *Link-State Service* is used by a node to query information about the state of its adjacent links or the state of the links of one of its neighbors. A link state is a triple $(x, y, state)$ where *state* is a record and x and y are the connection endpoints. In our implementation, the *state* includes information like the associated weight, whose estimation is explained later in this chapter, the type of the node, the power loss, and the time when the node was last seen. Note that our Link-State Service is not a fully fledged Link-State service as for example OSPF [Moy97], because it is a lot simpler and does not use flooding to disseminate information. It was designed according to the following requirements:

- If x is a node and y and z are in the vicinity of this node, then it must be possible for x to get the state of the connection (y, z). This requirement originates from the propose modules.

- It must be possible for a node x to estimate the number of currently alive neighbors. This is required to limit the number of proposals or messages, because their complexity can be in $O(n^k)$ [TM05, p.82]. Our first implementation did not use this information, and we found out that there is no suitable constant time for the triggering of the propose modules in any network. Or in other words if new proposals are searched in a fixed time interval this approach does not scale well and there exists no constant which matches all networks. Therefore we have chosen a dynamic approach based on the size of the network.

- The average number of messages should be reasonably low.

- It must not make use of any high level services, because the network topology is not known to this service and might not have been established at the point when this service is started. The highest layer it is allowed to use is the MAC layer.

The version presented in this work is the third variant we implemented during our study. The first variant was a simple neighbor discovery protocol using `HELLO` messages. On top of this framework, a link-state service was implemented, which used explicit messages to query the state of the links from its neighbors. This first version did not scale very well, and we worked on a second version which used a caching protocol to improve performance. Despite all our efforts the performance was terrible. The reason for this is that during the generation of proposals the algorithm needs to know all possible group internal connections to calculate the best possible group. If a group has k members and the members themselves are again groups with k members, then there are k^2 terminal nodes to choose from. If we need all possible connections for our calculation, we need to query $\binom{k^2}{2} - k \cdot \binom{k}{2}$ connections. The last part in our equation is a correction factor and stems from the fact that we do not need to query the connections internal to a group because they are not needed to build the new group. In any case the complexity is in $O(k^4)$.

Therefore our third version follows a completely different approach and tries to reduce the number of messages exchanged. The first observation we can make is that to know that a node is alive, it is necessary that every node shows a sign of life. This can be done in a lot of different ways, but in our framework we limit exchange of information to information disposal by messages[4]. Therefore every node must transmit a message periodically, which implies that any algorithm following this approach must be at least in $O(n)$. Having this information allows a node to estimate the number of currently alive neighbors. Note that this is still not enough for our requirement, because a node must associate a connection with a weight and should also be able to know the connections between two other neighbors in its vicinity. Therefore we decided to piggy-back additional information within the messages. Every message contains the power level used for transmission and additional information, so called *flags*. Depending on the type of message used, additional information about neighbors is also included. The reason why we decided on this is that it does not make much difference if a bigger frame is sent instead of a small one, because a lot of overhead in IEEE802.11 wireless networks originates from the carrier sense algorithms and back off times. The different type of messages are briefly described below:

Full Update: A full update transmits the complete list of neighbors at a node. A node receiving this information adds it to its own database and marks all entries with the timestamp when the message was received. If periodic `HELLO` messages are sent every x seconds, then the information at this node is a most $2x$ seconds old[5].

Partial Update: A partial update transmits only the modifications between the last full update. Partial updates have a lower message size and can be enabled if the network topology changes often. They are used to update the state information on some nodes. For example a node could be marked as unreachable or its weight could be updated.

To allow late joining of a node, it is necessary to transmit full updates periodically. For this purpose the constant `LSB_FULL_UPDATE_INTERVAL` is set to 5. If the periodic hello interval `LSB_HELLO_INTERVAL` is set to 5 seconds this would imply that every 25 seconds a full update message is broadcasted to all neighbors. During the other times only partial updates are sent.

Definition 30 (Link-state record)**.** *A* Link-state record *is a triple* $(x, y, state)$ *where* $x, y \in V$ *and state is record like type as shown in Listing 3.8:*

Listing 3.8: Link-State state

```
Record state is {
  Weight      : Weight of the connection. Weight ∈ ℝ+.
```

[4] This means that there is not other type of communication between nodes (for example visibility, noise, ...).

[5] Because the neighbor estimate from the other node is also only updated every x seconds and the information at the receiving node remains valid for up to x seconds.

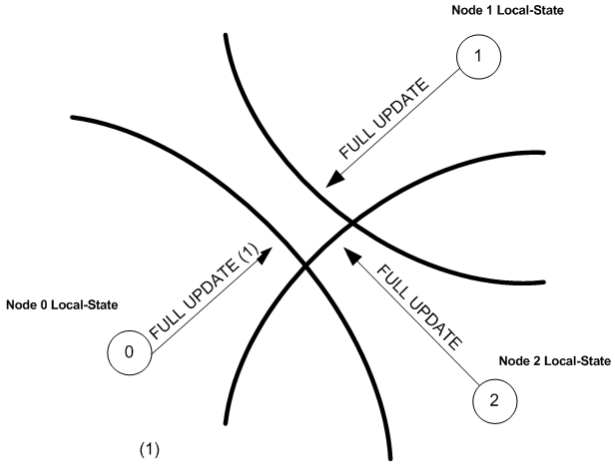
Lastseen : *Time when this information was last updated. Lastseen* $\in \mathbb{R}+$
Flags : *Additional information about this node. The list of possible flags is shown in Table 3.7*
}

Table 3.7: Flags for a Link-State state

Flags	*Value*	*Description*
IS_GATEWAY	0x01	This node is a Gateway node.
IS_DOWN	0x02	This node is down.

Example

An example of the basic operational principle of the algorithm is shown in this section. We start with three nodes 0, 1 and 2. Furthermore, we assume that no information is available directly after startup. New nodes start by sending a FULL_UPDATE message initially. This is shown in Figure 3.11. Note that the times when the messages are sent are not correlated.



MAC header				Frame body					CRC
...	Addr1	Addr1	...	Type	Pwr	Flags	Fr.	#Fr.	...
...	OR	FF	...	0x00	25[dBm]	0x00	0	1	...

Figure 3.11: Sending of FULL_UPDATE message immediately after startup.

Now assume that some time has passed and all nodes have received the frame shown in this example. The details of the MAC frame are explained later in this chapter. Therefore every node now knows some information about its neighbors. This is shown in Figure 3.12.

In the next FULL_UPDATE message the new information is added as piggy-back data to the message. Therefore the message from nodes 0 includes two additional link states, where $State[0]$ includes the address of node 1, its weight 63 and node $1'$s flags. The state

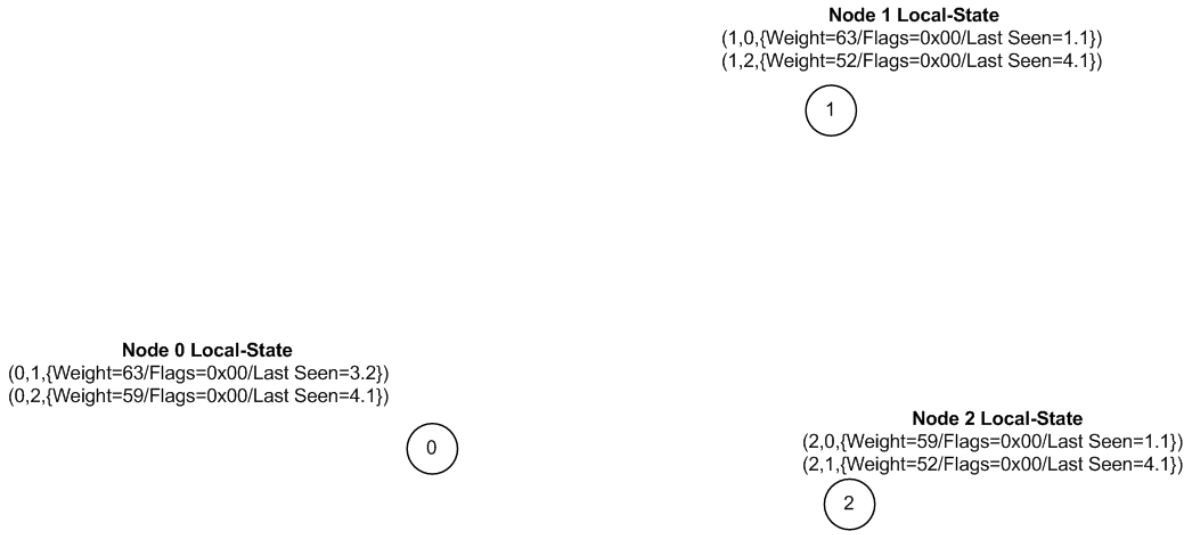
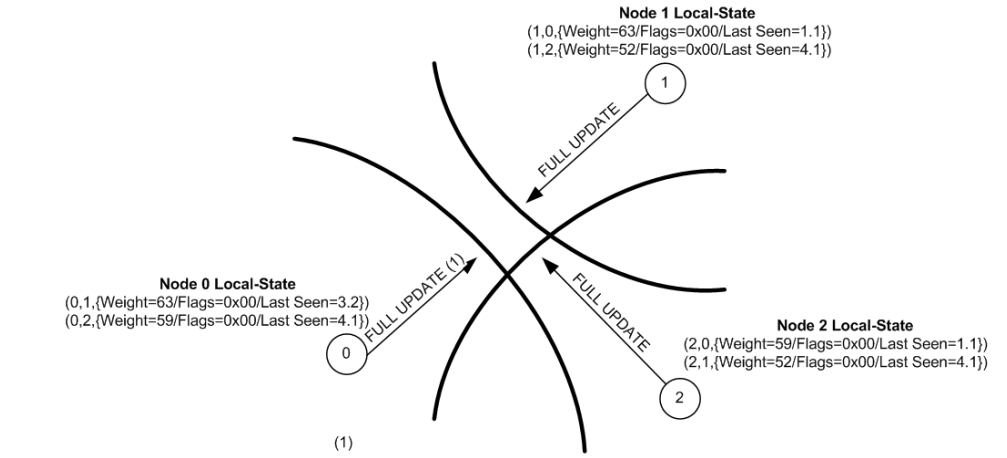


Figure 3.12: Link-state database after reception of the first `FULL_UPDATE` messages.

for node 2 is transmitted in $State[1]$. This is shown in Figure 3.13.

The final situation is shown in Figure 3.14. Having this information, the local propose module (and almost all others) has all required information available to calculate their proposals. It is obvious that this algorithm is in $O(n)$, and for medium sized networks a single message is sufficient to broadcast all the required information. If only a single query has to be performed by any proposals, this algorithm already gives better results than one using explicit messages. Internally, the information is stored within a hash map to allow fast and efficient lookups.



MAC header				Frame body												CRC
...	Addr1	Addr1	...	Type	Pwr	Flags	Fr.	#Fr.	States	State[0]			State[1]			...
...	OR	FF	...	0x00	25[dBm]	0x00	0	1	2	1	63	0x00	2	59	0x00	...

Figure 3.13: Second `FULL_UPDATE` message with piggy-back data.

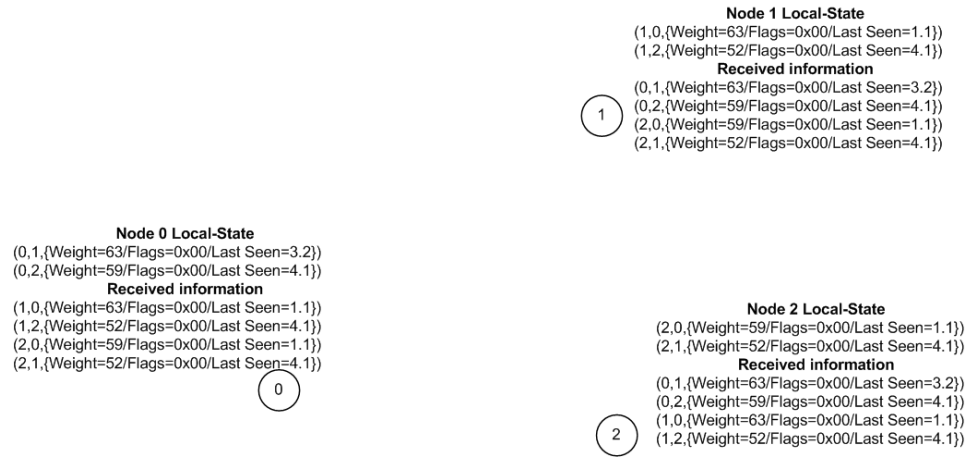


Figure 3.14: Link-state database after reception of the second `FULL_UPDATE` messages.

Frame Format

There are two different types of messages. The full update message is shown in Table 3.8, where the link state payload is composed of objects from the type shown in Table 3.9. The full update messages allow up to 256 nodes in the data payload, resulting in a total size of the IEEE 802.11 frame body of $2053B$, which is less than the maximum of $2312B$. If more than 256 neighbors are available, the frame numbers can be used to fragment the message.

Table 3.8: Link-state `FULL_UPDATE` message

MAC header				Frame Body								CRC
...	*Addr1*	*Addr2*	...	*Type*	*Pwr*	*Flags*	*Fr.*	*#Fr.*	*States*	*State[0]*	...	*State[k-1]*	...
...	`OR`	`FF`	...	`0x00`	`0xXX`	`0xXX`	i	$\#Fr.$	k	...	...	...	...

OR ... `OR:OR:OR:OR:OR:OR` is the source MAC address.
FF ... `FF:FF:FF:FF:FF:FF` is the MAC broadcast address.
Type ... Unsigned 8Bit quantity set to `LSB_FULLUPDATE_MSGTYPE = 0`.
Pwr ... Transmission power in dBm as signed 8Bit quantity ($-128 - +127[dBm]$).
Flags ... Flags for the node which sent this message.
Fr. ... Frame number i of $\#Fr$ as unsigned 8Bit quantity starting at 0. Used when the number of neighbors exceeds 256.
#Fr. ... Total number of Frames as unsigned 8Bit quantity.
States ... Unsigned 8Bit quantity set to the number of Link-State entries in this message.
State[k] ... An entry in the format described in Table 3.9.

Table 3.9: Link-state field within a message.

Neighbor address (6Byte)	*Weight (4Byte)*	*Flags (1Byte)*
`XX:XX:XX:XX:XX:XX`	Weight	0xXX

Weight ... A IEEE754 floating point number representing the weight for the connection.
Flags ... OR-Combination of flags defined in 3.7.

The partial update messages are shown in Table 3.10. Any number of partial update messages can be sent between full update messages, but their number should be kept low to reduce network load.

Table 3.10: Link-State Partial Update message

MAC header				Frame Body							CRC
...	*Addr1*	*Addr2*	...	*Type*	*Pwr*	*Flags*	*States*	*State[0]*	...	*State[k-1]*	...
...	`OR`	`FF`	...	`0x00`	`0xXX`	`0xXX`	k	...	...	...	...

Type ... Unsigned 8Bit quantity set to `LSB_PARTIAL_UPDATE_MSGTYPE = 1`.
See Table 3.8 for a description of the other fields.

3.6.5 Weight Estimation

What we have left out in our discussion up to know is how the weight are estimated. A very natural approach for wireless networks is to use the path loss as an estimate for the cost between nodes. We have chosen a very basic model of our real world environment and assume a free-space model with no interference. Then the following equation can be used to describe the received power at a node.

$$P_r = O\left(\frac{P_t}{d^\alpha}\right)$$

Here P_r is the received power in Watts, P_t is the transmission power used by the transmitting node and α is an abstraction parameter. A more detailed discussion on these equations has already been given in Section 2.1. Converting the values to dBm[6] gives us the following equation

$$10\log\left(\frac{P_r}{1mW}\right) = 10\log\left(\frac{K\left(\frac{P_t}{d^\alpha}\right)}{1mW}\right)$$

$$P_{r[dBm]} = 10\log K + 10\log\left(\frac{P_t}{1mW}\right) - 10\alpha\log d$$

$$P_{r[dBm]} = 10\log K + P_{t[dBm]} - 10\alpha\log d$$

Now we can deduce the following equation if we ignore constant terms.

$$\log d = \frac{P_{r[dBm]} - 10\log K - P_{t[dBm]}}{-10\alpha}$$

$$\log d = \frac{P_{t[dBm]} - P_{r[dBm]} + 10\log K}{10\alpha}$$

$$\omega = \log d = O\left(P_{t[dBm]} - P_{r[dBm]}\right)$$

For example in the typical NS2 scenario with a transmission power $P_t = 0.281W = 24.5dBm$ and a receive threshold of $R_x = 3.652 \cdot 10^{-10}W = -64.4dBm$ the maximum weight is $\omega = 88.9$. Higher values would result in a node not being recognized as reachable. Now that we have a basic weight estimate, we can use this algorithm to implement our weight definition shown in Definition 10. For connections between normal nodes, this is already sufficient, but connections between gateway nodes and gateway and normal nodes must be treated differently.

Definition 31. *Let P_t and P_r be power levels in dBm. Then the weight between two nodes is defined as.*

$$\omega = \begin{cases} P_t - P_r & \text{iff } x, y \in V' \\ 2k^2 512 + (P_t - P_r) = 1024k^2 + P_t - P_r) & \text{iff } \begin{matrix}(x \in V' \wedge y \in V'') \\ (x \in V'' \wedge y \in V')\end{matrix} \vee \\ 2k^2(2k^2 512) + (P_t - P_r) = 4096k^4 + (P_t - P_r) = & \text{iff } x, y \in V'' \end{cases}$$

The factors before the path loss are based on the weight condition defined by Thallner in [TM05, p9]. The 512 is the maximum weight, which can result by the subtraction of the two 8-bit integer values which are used to hold the transmission and receive power levels[7]. For the gateway to gateway connections, the same formula was applied again, but the weight chosen is the weight from the next lower groups. This weight is approx. $2k^2 \cdot 512$ which is the maximum weight of any connection between normal and gateway nodes.

[6] dBm is a logarithmic measurement for the power level and the dBm value of any power in Watts is given by $P_{dBm} = 10\log_{10}(P_{Watts}/1mW)$.

[7] Actually 256 would suffice ,because the integers are signed, but the exact number does not matter as long as it is larger than the maximum weight.

Interface

The basic operation of the link-state service has already been described. We will now focus on the important implementation details and will show how the link-state service can be used. The most important operations for any link-state service are shown in Figure 3.15. The class LinkState supports the retrieval of connection information using the methods LinkState::query_all_edges and LinkStateService::query_edges. The former one obtains a list of all connections, where as the latter one supports a filter which is beneficial if a node has a large number of neighbors. Furthermore, it supports methods for getting a list of neighbors with their connections (LinkState::neighbor_connections), a list of neighbors (LinkState::neighbors, and methods for testing if a given node is a neighbor.

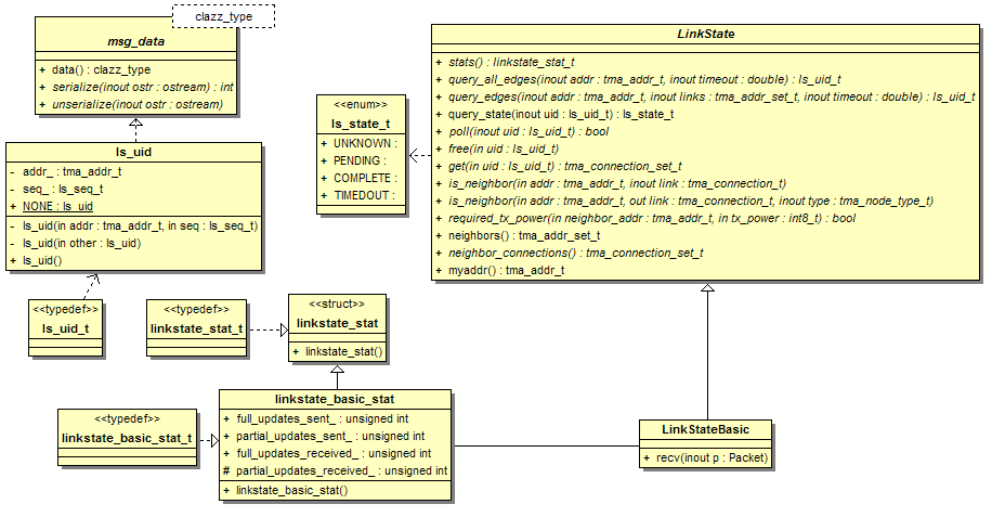


Figure 3.15: UML class diagram for Link State Service

Because method calls within NS2 must not block and the answer from the remote node is not immediately available, communication must be broken down into multiple states. The link-state service associates each query with a unique identifier of type ls_uid_t . Such an identifier is returned when a query is executed. Using this identifier a node can check the state of a query which is either LinkState::COMPLETE, LinkState ::PENDING or LinkState::TIMEDOUT. State transistions are made upon message receival or an internal timer used to implement timeouts. If a query is in the state LinkState::COMPLETE, the retrieved set of edges can be obtained by calling LinkState ::get. In any case, a completed or timedout query should be freed by calling LinkState ::free.

An example which obtains the complete neighbor information for node 3 is shown in Listing 3.9. The function 'query_once' initiates a query. After a query has been started, the method 'check' should be called. If the query has been completed, the results are returned or errors are signaled. The method recv is used to dispatch network messages

for the link-state service.

Listing 3.9: Example for using the link-state service

```
1  class BasicLinkStateServiceTest : public BiConnector
2  {
3  private:
4    LinkState *ls_;
5    ls_uid_t my_query_;
6
7  protected:
8    void query_once( void )
9    {
10     tma_addr_t interesting_neighbor = 3;
11     // query neighbor 3 with a timeout of 10 seconds.
12     my_query_ = ls_->query_all_edges( interesting_neighbor, 10.0 );
13   }
14
15   void check( void )
16   {
17     switch ( ls_->query_state( my_query_ ) )
18     {
19     case LinkState::PENDING:
20       break;
21     case LinkState::COMPLETE:
22     {
23       tma_connection_set_t connections = ls_->get( my_query_ );
24       // use the connections
25       ls_->free( my_query_ );
26     }
27     break;
28     case LinkState::TIMEDOUT:
29       // Other node did not answer. Report an error and free resources.
30       ls_->free( my_query_ );
31       break;
32     }
33   }
34
35 public:
36   virtual void recv( Packet *p, Handler *cb )
37   {
38     hdr_cmn *hdr = hdr_cmn::access( p );
39
40     // Process any multicast packets.
41     if ( hdr->ptype( ) == PT_LSB )
42     {
43       if (( ls_ != NULL ) && ( typeid( LinkStateBasic ) == typeid( *ls_ ) )
             )
44       {
45         dynamic_cast<LinkStateBasic *>( ls_ )->recv( p );
46       }
47     }
48   };
49 };
```

3.6.6 Non-Blocking Atomic Commitment

The *non-blocking atomic commitment (NBAC)* service is implemented in the C++ class NBAC. If a module wants to use the NBAC protocol, it has to subclass the NBAC class and provide implementations for the NBAC::vote, NBAC::decision, and if required, for the NBAC::finalize methods. The basic methods of the NBAC class are shown in Figure 3.16.

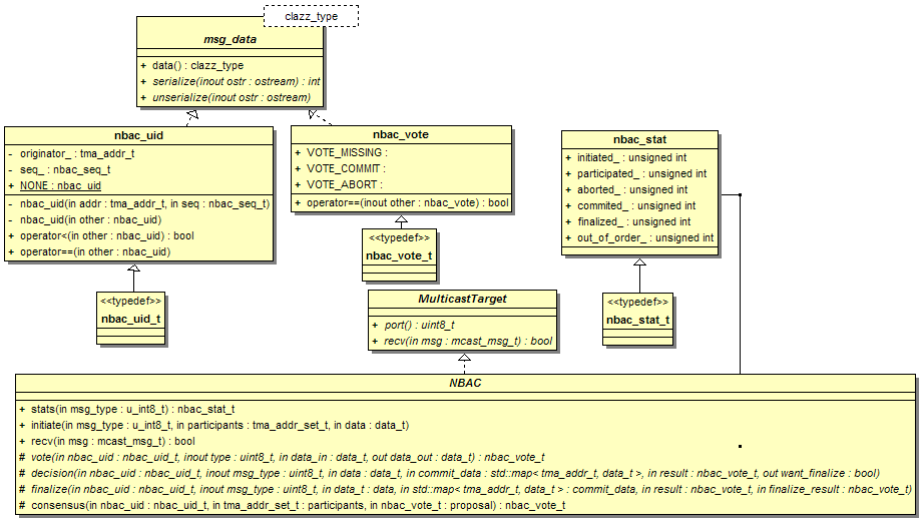


Figure 3.16: UML class diagram for Non-Blocking Atomic Commitment

The only method required to initiate an NBAC phase is to call NBAC::initiate, where three arguments have to be provided. The first one, msg_type, is used to distinguish between different application dependent NBAC types[8]. The second one, participants, is a set of participants which participate in the NBAC protocol. The last argument, data, is the data, which is initially voted upon. An actual implementation is given in NBACThallner in the file `tma-thallner.cc`.

Basic operation

In NS2 it is not possible to block on the reception of a message within the code. Therefore, the NBAC protocol has to be split into multiple states. The transition from states happens through the use of timeouts and the reception of new multicast messages. The basic phases are shown in Figure 3.17. If the initiator calls NBAC::initiate, it sends a message of type `NBAC_MESSAGE_TYPE_INITIATE` to all participants (including itself). Then every node votes upon the data by calling NBAC::vote. These votes are then multicasted as `NBAC_MESSAGE_TYPE_VOTE` messages to all other participants. It is also possible for this type of messages to contain so called piggy-back data. Because our reliable multicasting service guarantees delivery, every node will receive all other votes. If

[8] For example, in our case the group checking and the proposal of new groups.

all votes have been received or a node has been suspected (not implemented) every node calls the consensus protocol[9]. The proposed value is nbac_vote_t :: VOTE_COMMIT, if all received votes are nbac_vote_t :: VOTE_COMMIT. In all other cases, the vote is nbac_vote_t :: VOTE_ABORT. After the consensus, every node calls NBAC::decision with the agreed values. If a finalization is needed, another step is executed. For the finalization step, every node which has called NBAC::decision sends a multicast message of type `NBAC_MESSAGE_TYPE_FINALIZE` to all other nodes. If every node has received this message, they can be sure that every node has executed NBAC::decision. The goal of finalization is to dissemniate the information that consensus has been completed on all participants.

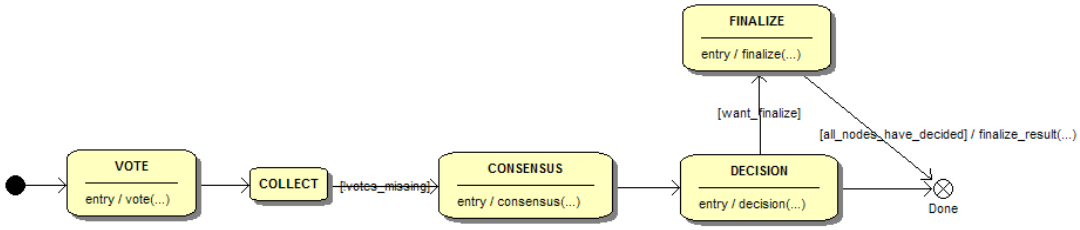


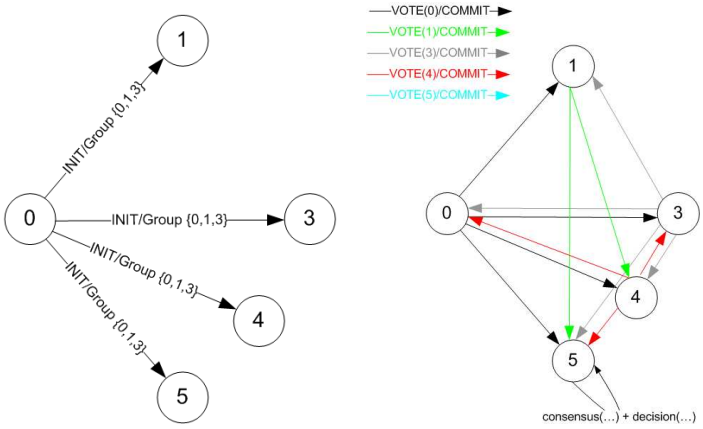
Figure 3.17: State diagram for NBAC implementation

For better illustration, we have also shown the basic execution in a set of diagrams in Figure 3.18. Please note that every diagram contains a large number of individual computation steps at any node, but it should show the basic operation of the protocol. In this example node 0 initiates a NBAC with a new group proposal for the group $\{0, 1, 3\}$ with members $\{\{0\}, \{1\}, \{3, 4, 5\}\}$. In 3.18(a) we can see that node 0 has initiated the NBAC. In 3.18(b) we see that all nodes receive and exchange votes. After a node has received all votes, like node number 5, it starts executing the consensus protocol and finally calls NBAC::decision. The group construction requires a finalize step, where every node sends a finalized message immediately after calling decision. This is shown in 3.18(c) for nodes $0, 1$ and 5. After a node has received all finalize messages, it executes NBAC::finalize and the protocol has finished.

Frame formats

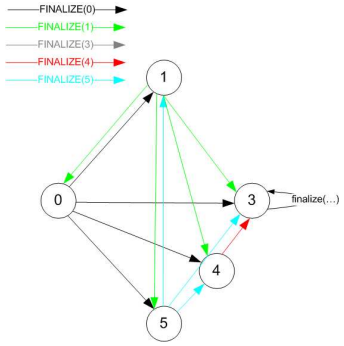
The NBAC protocol uses three different kind of messages. Messages of type `NBAC_MESSAGE_TYPE_INITIATE` are shown in Table 3.11. They contain the data payload which is initially voted upon. This is the first message sent. The vote responses are shown in Table 3.12; they can contain a data payload, also called *commit data*. The `NBAC_MESSAGE_TYPE_FINALIZE` are very simple messages and shown in Table 3.13; They are only used for synchronization.

[9] Which in our case is very simple, because we do not allow any nodes to fail. Furthermore, a reliable transport is always required, because there exists an impossibility result for NBAC with unreliable



(a) Node 0 initiates group proposal for $\{0, 1, 3\}$

(b) Nodes voting and deciding on NBAC outcome



(c) Finalizing NBAC

Figure 3.18: Example execution of NBAC protocol for a group proposal

NBAC UID (10B)		Type (1B)	App. Type (1B)	Payload
`OR:OR:OR:OR:OR:OR`	`SEQ`	`NBAC_MESSAGE_TYPE_INITIATE=1`	`0xXX`	...

OR ... NBAC initiator
Application Type ... Used for protocol multiplexing.

Table 3.11: Format of NBAC initiate message

NBAC UID (10B)		Type (1B)	App. Type (1B)	Vote (4B)	Payload
`OR:OR:OR:OR:OR:OR`	`SEQ`	`NBAC_MESSAGE_TYPE_VOTE=2`	`0xXX`	Vote	...

OR ... NBAC initiator
Application Type ... Used for protocol multiplexing.
Payload ... Commit data.
Vote ... Either `VOTE_COMMIT=1` or `VOTE_ABORT=2`.

Table 3.12: Format of NBAC vote message

NBAC UID (10B)		Type (1B)	Application Type (1B)
OR:OR:OR:OR:OR:OR	SEQ	NBAC_MESSAGE_TYPE_FINALIZE=3	0xXX

OR . . . NBAC initiator
Application Type . . . Used for protocol multiplexing.

Table 3.13: Format of NBAC finalize message

links [Gra78].

3.7 Group Checking and Group Construction

3.7.1 Datatypes and classes

The most important data types are shown in the UML diagram 3.19. These mappings correspond exactly to the records defined in Section 2.3.1. In addition the most

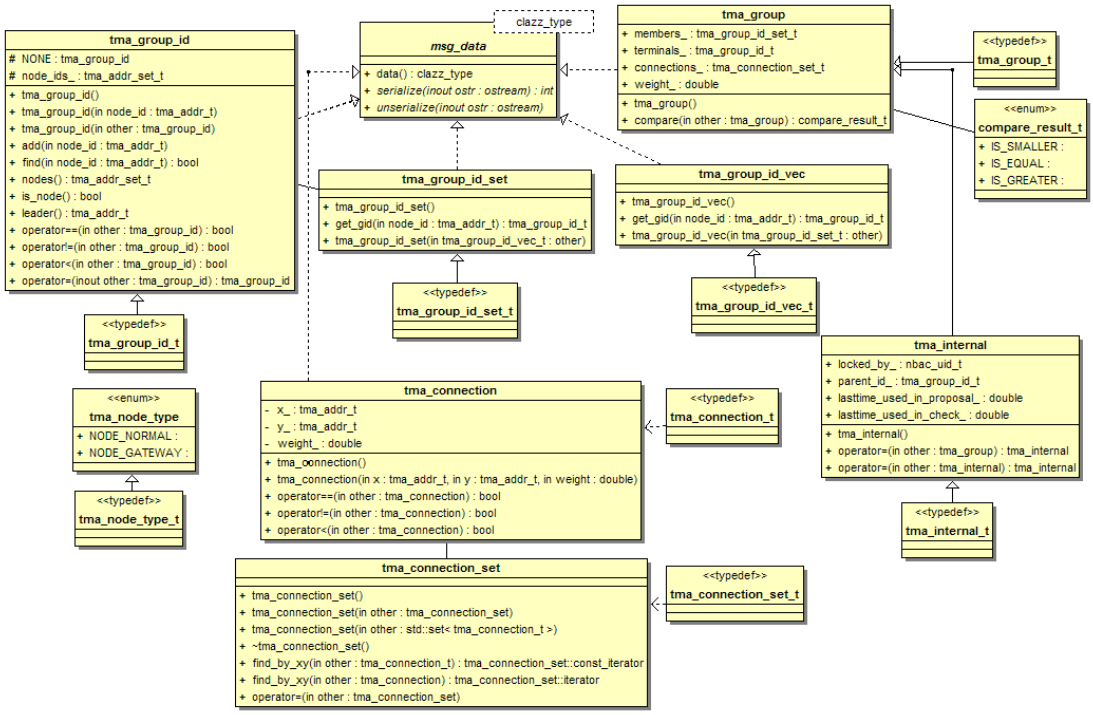


Figure 3.19: UML class diagram for basic datatypes used by the TMA

important methods and attributes of the Thallner algorithm are shown in Figure 3.20. The actual implementation is in the files `tma-thallner.cc` and `tma-thallner.h`, where most of the methods are simply C++ equivalents of the algorithms shown in 2.3. Newly introduced functions are 'check_groups' and ' check_groups_initiate '. The first method estimates the suggested time between group checking. This time is subject to network size, network load, and the value of k. The exact calculation is shown in Section 3.7.2. Within this time interval, a node randomly initiates a group checking instance by calling ' check_groups_initiate '. This prevents node synchronization and makes sure that group checking is fair, because every node is at least triggered once within this interval, which also simplifies convergence algorithms. The time windows is cut into slices of the estimated time between proposals. Within these time slices a random instant is chosen where the group checking is performed. The functions 'propose_groups' and 'propose_groups_initiate' follow the same concept.

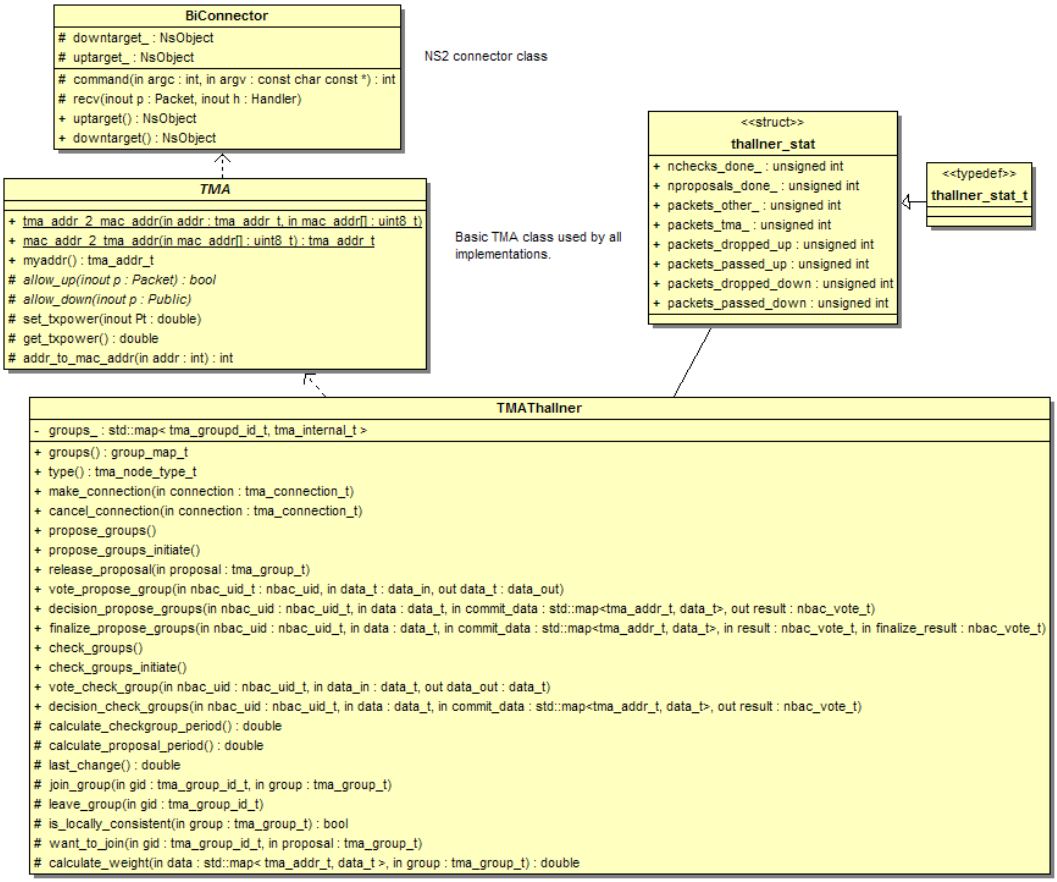


Figure 3.20: UML class diagram for TMA

3.7.2 Group checking

As already shown in Listing 2.4 in lines 24−30, the Thallner algorithm performs periodic group checking. Group checking serves the following purposes [TM05, p.39]:

- It removes broken groups, which can occur when a node leaves a group.
- It detects node crashes.
- The group weight adapts to changed connection weights.

We have already mentioned that the group checking is triggered by a periodic signal. The basic relationship is shown in Figure 3.21. We can see that the function 'check_groups' is called at constant time intervals. This time is calculated by the function 'calculate_checkgroup_period'. Within this fixed time window, group checking is initiated by calling the function ' check_groups_initiate '. The timers are realized by two

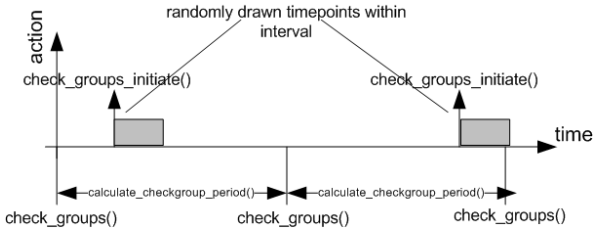


Figure 3.21: Periodic checking of groups

instances of the generic timer class GenericTimer shown in Figure 3.22. Such a timer instance requires an object, which is parameterized by the template typename 'agent'[10], and a function pointer to a method of this class. If the timer has expired, the function arg_fp. The source code of the method 'check_groups' is shown in Listing 3.10 with the debugging code removed.

Listing 3.10: C++ group checking code

```
1  void
2  TMAThallner::check_groups( void )
3  {
4    // calculate basic time slice
5    double period = calculate_checkgroup_period( );
6    // calculate time when the actual check is scheduled
7    double scheduled_time = Random::uniform( ) * period;
8    // schedule the timers
9    chkgrp_tmr_real_.resched( scheduled_time );
10   chkgrp_tmr_.resched( period );}
11 }
```

[10] An agent is NS2 is an object which sends and receive messages and we have chosen to use this name. But of course any other class can be used.

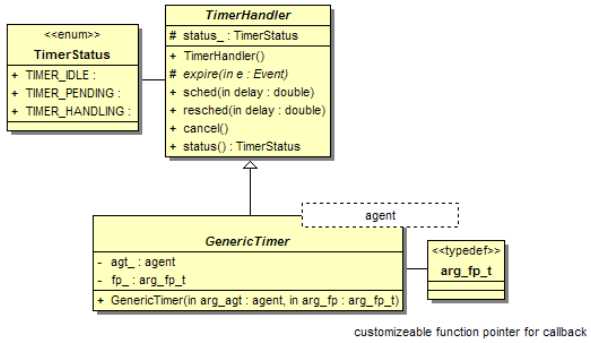


Figure 3.22: UML class diagram for generic timer

The real work is done by the function 'TMAThallner::check_groups_initiate' which is shown in Listing 3.11. Again we have removed some debug code to ease the presentation. We start by recursively climbing the group hierarchy in the while loop shown in lines $4 - 24$. If the groups are locked because they are currently under construction, we abort our search. This check if performed in line 7. From all available groups in the hierarchy, the group which has not been checked recently, is chosen in line 14. This check is implemented by storing the time when the group was checked in the group internal data structures. This is necessary because if all groups were checked at once, network congestion would result if the total number of groups (which is $O(n)$) becomes bigger. Using this implementation, it is guaranteed that only one group is checked within a given time interval. Since all groups are checked eventually, the theoretical results for the resulting topology graph are not affected, although the time complexity is increased[11]. The actual check is performed in lines $27 - 32$. We can see that first the time is updated in line 29, then a new NBAC instance is created in line 30, and finally the statistical information is updated in line 31.

Listing 3.11: C++ group checking code

```
1
2   tma_group_id_t current_gid = groups_[ tma_group_id_t( myaddr( ) )].
        parent_id_;
3   tma_group_id_t final_gid = tma_group_id_t::NONE;
4   while ( tma_group_id_t::NONE != current_gid )
5   {
6     // Check that the group data structures are not locked.
7     if ( nbac_uid_t::NONE == groups_[ current_gid ].locked_by_ )
8     {
9       if ( tma_group_id_t::NONE == final_gid )
10      {
11        final_gid = current_gid;
```

[11] For very small network sizes it is possible to disable this algorithm but this is not recommended by the author.

```
12        }
13        // Find the group which has not been checked recently.
14        if ( groups_[ current_gid ].lasttime_used_in_check_ < groups_[
              final_gid ].lasttime_used_in_check_ )
15        {
16          final_gid = current_gid;
17        }
18        current_gid = groups_[ current_gid ].parent_id_;
19      }
20      else
21      {
22        break;
23      }
24    }
25
26    // If a candidate group has been found for checking then do it.
27    if ( tma_group_id_t::NONE != final_gid )
28    {
29      groups_[ final_gid ].lasttime_used_in_check_ = Scheduler::instance( ).
            clock( );
30      nbac_->initiate_check_group( groups_[ final_gid ] );
31      stats_.nchecks_done_++;
32    }
```

To complete the group checking algorithm we also have to implement the functions required by the NBAC service defined in Section 3.6.6. Therefore we have subclassed the NBAC class and created a new class NBACThallner, which is shown in Figure 3.23. The method 'NBACThallner::initiate_check_group' takes a group, serializes the complete group data structure, and initiates a NBAC with all terminal nodes of the group members. The method 'NBACThallner::decision' demultiplexes the NBAC instances, and if the NBAC message is of type `NBAC_TYPE_CHECK_GROUP`, it is dispatched to 'TMAThallner::decision_check_groups'. The same holds for the 'NBACThallner::vote ', which is dispatched to 'TMAThallner::vote'. Group checking needs no finalization and therefore finalization can be left out. Actual implementations of the functions are available in `tma-thallner.cc` and are not repeated here because they are simply C++ equivalents of the algorithms shown in 2.3.

Periodic triggering

The group checking algorithm has a message complexity of $O(k^4)$: A node initiates the NBAC and therefore broadcasts a message. There are up to k^2 participants, and because of the reliable multicast, this gives us $1 + (k^2 - 1)$ messages[12]. Now every node broadcasts its vote. For every broadcast, all members must answer with an acknowledge because of the reliable multicast. Having k^2 nodes, this gives us $k^2 \cdot (k^2 - 1)$ messages. Therefore the message complexity is in $O(k^4)$. The total number of groups in a graph is given by $\left\lfloor \frac{n-1}{k-1} \right\rfloor$ (See Section 2.1.3). Combining these theoretical values we arrived at the following formula for the period of the group checks.

[12]The -1 is because the initiator must not acknowledge its message.

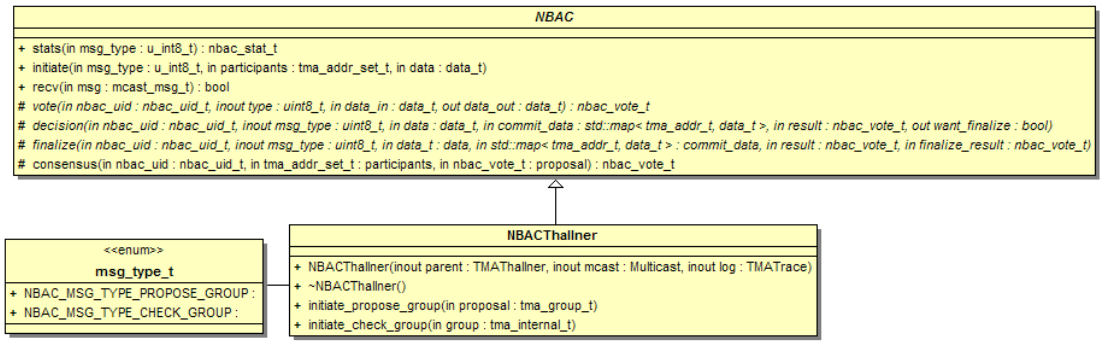


Figure 3.23: UML class diagram for Thaller NBAC

$$t_{CheckgroupPeriod} = 0.001 \times k^4 \times \frac{1.0}{\texttt{CHECKGROUP\_NETWORK\_LOAD}}$$

The factor 0.001 was obtained empirically by using simulation results.

3.7.3 Group proposals

Group proposal are created by triggering the propose module, which then performs a search. The actual implementation of the propose module does not matter, as long as it supports or common interface, but if necessary we will refer to our local propose module from Section 3.8. During our work, we tried to use a periodic trigger for the generation of new proposals. Obviously, this approach cannot scale very well, because it does not know anything about the network. It was first set to 5 seconds, which might seem quite large, but it already had problems with network sizes exceeding 20 nodes. This is especially problematic because some algorithms are in $O(n^n)$. Because of these problems, we dropped our initial approach and decided to implement an approach which dynamically adapts to the network size. The basic algorithm is as follows:

1. Every node estimates the number of neighbors by querying the link state service. The link state service has this information because of the underlying neighbor discovery protocol.

2. Every node has an estimate on how long it will take to generate a new proposal. This estimate is obtained from the propose module and can either be a constant or a dynamic value. See 'TMALNPProposeModule::estimated_proposal_time' in '`tma-lnp-proposal.cc`' for a concrete implementation. For example, for the local non-perfect propose module this value, is computed by $t_{EstimatedTime} = \texttt{MESSAGE\_CONSTANT} \times n^2$. The last part of this equation is based on the average message complexity of [TM05, p. 82] which is in $O(n^2)$. The first constant was obtained using empirical results obtained by simulation.

3. Every node reschedules one proposal generation within the time frame defined by

$$t_{ProposePeriod} = t_{EstimatedTime} \times n \times \frac{1.0}{\texttt{PROPOSE\_NETWORK\_LOAD\_AVG}}$$

Here $t_{EstimatedTime} \times n$ is the time it takes to generate proposals on all nodes. The last term, using the constant `PROPOSE_NETWORK_LOAD_AVG`, is used to control the network load. It is important that nodes try to reschedule their proposal generation randomly within this timeframe. The value of n is important because every node wants to initiate a search.

Note that the proposal period time can also be used for network convergence detection. If within a given time frame no proposals changed the network topology, it can be safely assumed that the network is stable. The time frame is given by $c \times t_{ProposePeriod}$, where c depends on the type of propose module. For a Local-Non-Perfect Propose Module, this is set to $c = 2.0 \cdot t_{ProposePeriod} \cdot \log_k \left(\lfloor \frac{n-1}{k-1} \rfloor \right)$, which is proportional to the average depth of a balanced tree for groups of size k if the time $t_{ProposePeriod}$ is assumed as constant within this context.

The basic implementation follows the same principles as the group checking. The method 'TMAThallner::propose_group' is called with the estimated period. Within this time a random instant is chosen, where the actual search is initiated. This is shown in Figure 3.24, and the actual implementation is shown in Listing 3.12.

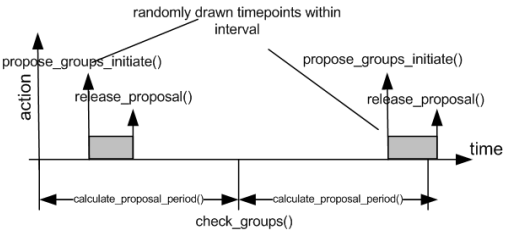


Figure 3.24: Periodic triggering of propose module

Listing 3.12: Triggering of group proposals in C++

```
1  void
2  TMAThallner::propose_groups( void )
3  {
4    int number_neighbors = ls_->neighbors( ).size( );
5
6    // It does not make any sense to generate proposals if there are not
7    // enough neighbors
8    double period;
9    if ( number_neighbors > THALLNER_K )
10   {
11     period = calculate_proposal_period( );
12
13     // When to generate a proposal within this time window.
```

```
14      double scheduled_time = Random::uniform( ) * period;
15      propgrp_tmr_real_.resched( scheduled_time );
16    }
17    // We can't do anything up to now. Try again in one second.
18    else
19    {
20      period = 1.0;
21    }
22    propgrp_tmr_.resched( period );
23  }
```

The real work is done in the function 'TMAThallner::propose_groups_initiate' shown in Listing 3.13. This method is designed for the local propose module and periodically initiates new searches. Therefore it is necessary to recursively walk up the group hierarchy at the node and try to find new proposals containing this group id. The reason for this is that a local propose module by definition always includes the group id of the proposer [TM05, p. 60]. This is implemented in lines $8 - 27$. A node will only initiate searches for which it is a leader, which is tested in line 10. From all possible groups, the group which has not been used for the longest time is selected. Again this is done by storing additional information, which is simply the time when the proposal was last used. Finally, the search is started using the group found in line 31. Furthermore, the statistical information is updated in line 30, and the time when the proposals was last used is updated in 32.

Listing 3.13: Initiaing a group proposals in C++

```
1  void
2  TMAThallner::propose_groups_initiate( void )
3  {
4    // Recursively walk up the hierarchy and generate new proposals using the
5    // GIDs I am a leader in.
6    group_map_t::iterator current_gid = groups_.find( myaddr( ) );
7    group_map_t::iterator final_gid = current_gid;
8    while ( current_gid != groups_.end( ) )
9    {
10     if ( current_gid->first.leader( ) == myaddr( ) )
11     {
12       if ( current_gid->second.lasttime_used_in_proposal_ < final_gid->
             second.lasttime_used_in_proposal_ )
13       {
14         final_gid = current_gid;
15       }
16     }
17
18     // Check if this group has a parent. If yes climb the hierarchy.
19     if ( tma_group_id_t::NONE != current_gid->second.parent_id_ )
20     {
21       current_gid = groups_.find( current_gid->second.parent_id_ );
22     }
23     else
24     {
25       break;
```

```
26      }
27    }
28
29
30    stats_.nproposals_done_++;
31    prop_->local_non_perfect_proposal( final_gid->first );
32    final_gid->second.lasttime_used_in_proposal_ = Scheduler::instance().
          clock();
33  }
```

If a proposal has been generated by the propose module, the propose module must call the method 'TMAThallner::release_proposal'. This method initiates an atomic commitment with all members of the new group, and is shown in Listing 3.14. The NBAC service responsibility is the same as for the group checking, with the modification that messages of type `NBAC_MSG_TYPE_PROPOSE_GROUP` are dispatched to the appropriate methods 'NBACThallner::decision_propose_groups', 'NBACThallner::vote_propose_group', and 'NBACThallner::finalize'. Actual implementation of the functions are available in
`tma-thallner.cc` and are not repeated here, because they are simply C++ equivalents of the algorithms shown in Section 2.3.

Listing 3.14: Releasing of group proposals in C++

```
1  void
2  TMAThallner::release_proposal( const tma_group_t &proposal )
3  {
4    nbac_->initiate_propose_group( proposal );
5  }
```

3.8 Propose Module

We have provided a base class called TMAProposeModule which implements all the supporting functions for the propose modules defined in section 2.4.2. Its UML diagram is shown in Figure 3.25 and every new propose module should subclass from this abstract base class. Additional methods included in this class are TMAProposeModule ::estimated_proposal_time which must be provided by the actual implementation and TMAProposeModule::stats to get statistical data from the propose module. The basic idea behind this implementation is that the main algorithm periodically triggers the propose module to generate new proposals as shown in section 3.7.3. Then the propose module starts its algorithm and tries to find new proposals by any means. After it has found a (hopefully valid and useful) proposal it releases the proposal to the main algorithm by calling TMAThallner::release_proposal. The main algorithm takes this proposal and initiates the group construction algorithm described in section 3.7.3.

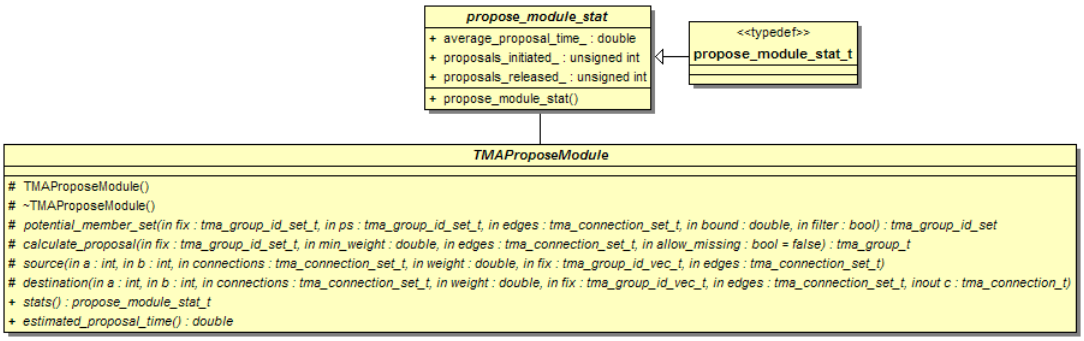


Figure 3.25: UML class diagram for TMA

3.8.1 Local non-perfect propose module

The implementation of the *local non-perfect propose module*, briefly LNP propose module, follows the description shown in section 2.4.3 with the following modifications:

- Every proposal contains a timestamp when the first SEARCH message was created. This can be used to estimate the average time needed to generate a proposals. Although it is implemented we have decided not to use it because it makes debugging more difficult if dynamic feedback is introduced into the algorithms. It can be enabled by setting the preprocessor flag PROPOSAL_ATTACH_CREATION_TIME in the file `tma-proposal.h` to either true or false. It increases the message size by 4 bytes.

- If enabled every proposal is tagged with a unique identifier. This is used to track proposals and is merely used for debugging purposes. It increases the message size by 8 bytes and is implemented by the class proposal_trace_id. It can

be enabled by setting the preprocessor flag PROPOSAL_ATTACH_ID in the file `tma-proposal.h`.

The UML class diagram for it is shown in Figure 3.26. Internally it uses the multicast service to send search messages to other nodes[13]. The Link-State service is used to query edges from other nodes because they are needed during the calculation. Furthermore it also requires a reference to the main instance of the Thallner algorithm to release proposals.

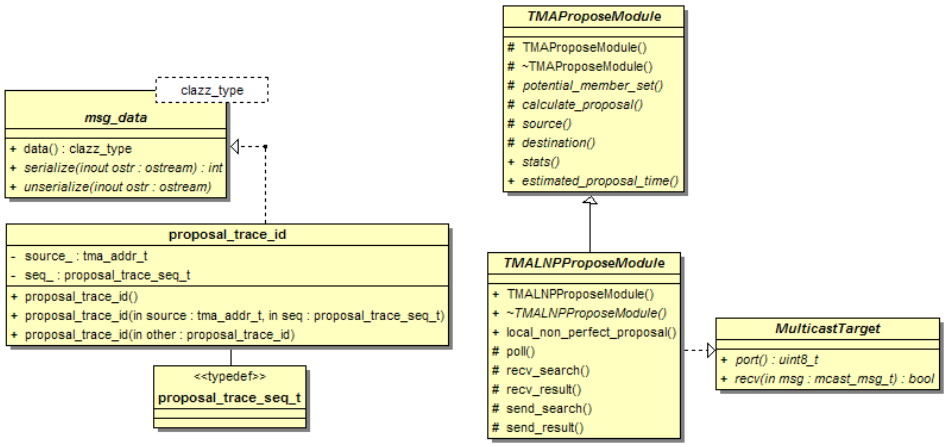


Figure 3.26: UML class diagram for TMA

A new search message is initiated by calling TMALNPProposeModule::local_non_perfect_proposal with an appropriate group identifier. This method corresponds to the function local_non_perfect_proposal shown in Listing 2.15 with the same name and creates a new TMALNPProposeModule::MSG_TYPE_SEARCH message which is sent to the leader of the group.

If a node receives such a search message the first time it creates a new state entry and the search is in its initial state INIT. The state transitions for a single search are shown in Figure 3.27. Reception of a message is implemented in the function TMALNPProposeModule::recv_search which is either called directly in case of a node local message or from TMALNPProposeModule::recv which handles the deserialization of the message first. In the INIT state all required edges for the proposal calculation are computed and link-state queries are sent. The query UIDs are added to the internal state and the state changes to QUERY. In the QUERY state it is checked periodically if the link-state queries are still pending, complete or have been aborted. If they are completed or an error has been detected this state is left.

If the queries are complete a transition is made to the state CALCULATE which matches exactly the function received shown in Listing 2.15 with the exception that the

[13] Note that our multicast service implementations are smart enough to use an unicast frame if there is only a single destination.

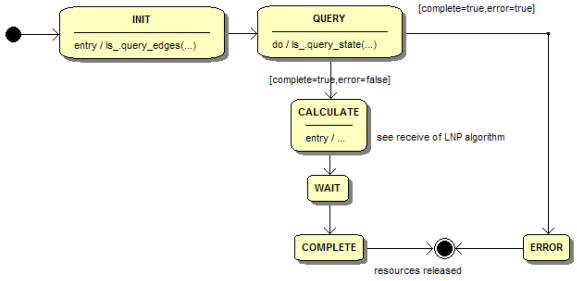


Figure 3.27: UML class diagram for TMA

query of the edges has already been performed. After the calculation a search moves to the states WAIT and COMPLETE where the resources are released. In case of an error a search immediately goes to the state ERROR.

In case a TMALNPProposeModule::MSG_TYPE_RESULT message is received it is released to the main Thallner instance by calling TMAThallner::release_proposal. Reception of such a message is implemented in recv_result which is either called directly or again from the TMALNPProposeModule::recv function which handles the deserialization.

4 Simulation

In this section we will show some simulation results to complete the theoretical results from Thallner shown in [TM05, p82]. If applicable, we will compare our results with that of Thallner.

4.1 Environment

Our simulation environment is based on NS2 with the components described in Chapter 3. The simulation environment provides the building blocks listed below. All filename references are relative to the `scripts/simulation` subdirectory unless otherwise noted.

- New random topologies can be generated by the script `adhoc-gen.tcl`. For example, invoking the command

  ```
  adhoc-gen.tcl 10 3 a
  ```

  creates a new topology under the subdirectory `topologies` for 10 nodes, 3 gateway nodes and is stored as variant a. The variant argument is used to allow multiple instances. The final result in the example above would be `topo-n10-gw3-a.tcl`.

- We use a convergence detection algorithm which monitors the time when a node has updated its group information. If all nodes have not changed their group information for a given amount of time, then the network is considered stable. This is explained in greater detail in the next section.

- Every node has a unique ID (based on its MAC address) and a position in the simulation area. Nodes can be accessed by the TCL variables node_(\$i) and their associated TMA instance tma(\$i). i is simply an index ranging over all available nodes.

- Nodes can move within the grid by standard means of NS2. A simple node movement for node 1 could be where node 1 starts moving to the x-coordinate 50.3 and the y-coordinate 70.8 at time 10.0 using a speed of $1.2 m/s$. This is shown in Listing 4.1.

  Listing 4.1: Node movement in NS2

  ```
  1 $ns at 10.0 1 setdest 50.3 70.8 1.2
  ```

- The weight of a node is dynamically calculated by the link-state service. Therefore, a node movement implies a change in the weights.

- We adapted the NS2 wireless settings to match the IEEE802.11b. This can be done by adding the settings shown in Listing 4.2 to the setup script[1].

[1] These settings are not very well documented, but seem to be an accepted standard for IEEE802.11b simulation accoriding to the NS2 users mailling list.

Listing 4.2: IEEE802.11b settings for NS2

```
1  Mac/802_11 set SlotTime_           0.000020        ;# 20us
2  Mac/802_11 set SIFS_               0.000010        ;# 10us
3  Mac/802_11 set PreambleLength_     144             ;# 144 bit
4  Mac/802_11 set PLCPHeaderLength_   48              ;# 48 bits
5  Mac/802_11 set PLCPDataRate_       1.0e6           ;# 1Mbps
6  Mac/802_11 set dataRate_           11.0e6          ;# 11Mbps
7  Mac/802_11 set basicRate_          1.0e6           ;# 1Mbps
8
9  Phy/WirelessPhy set freq_          2.4e+9          ;# Frequency
10 Phy/WirelessPhy set Pt_            3.3962527e-2    ;# Transmission
       power
11 Phy/WirelessPhy set RXThresh_      6.309573e-12    ;# Receiver
       threshold
12 Phy/WirelessPhy set CSThresh_      6.309573e-12    ;# Sense threshold
```

- Our simulation uses NS2 and therefore the computational model is based on the discrete event scheduler. To implement the periodic group checking and the group proposals, every node has multiple timers which trigger the computation steps. The period of the timers for the group checking module is calculated as described in Section 3.7.2. The calculation of the timer periods for the propose modules are shown in Section 3.7.3.

4.2 Simulation script

The simulation script is rather large and complicatem therefore we have decided to only discuss the most relevant parts in this section. All simulation scripts are available in the subdirectory `script/simulation`. The file `adhoc.tcl` is the main simulation script. The subfolder `topologies` holds all simulated network topologies, and the results are stored in the directories `nXX-gwY-[a-z]` where `XX` is the number of nodes, `Y` is the number of gateway nodes, and `[a-z]` is the topology variant. The toplogy variant is used to allow different network topologies for the same number of nodes. The simulation can be started by executing `batch.sh K ( XX)*` where `K` is the value of k and `XX` is the number of nodes. For example calling `batch.sh 3 5 10 15` would evaluate all network topologies for node sizes $5, 10$, and 15 for $k = 3$ by using NS2 and the main simulation script `adhoc.tcl`. The results would be stored in the subdirectories `n5-gw4-[a-z]`, `n10-gw4-[a-z]` and `n15-gw4-[a-z]`. The number of required gateway nodes is calculated automatically by the script. If a topology does not exist for a specific configuration, a new topology is created automatically by the script `adhoc-gen.tcl`. The generated topology is stored in `topologies/topo-nXX-gwY-[a-z]` using the same naming convention as above. The typical results for a single simulated topology are explained below.

4.2.1 Result files

Our simulation framework supports a lot of different outputs. All of them can be generated by calling the appropriate methods on the TMA objects, which are shown in

Section 3.5.2. The typical output for a simulation is shown in Listing 4.3, where the simulation was performed for 5 nodes with $k = 2$ and 2 gateway nodes.

Listing 4.3: Typical output files for a simulated topology

```
$ ls n5-gw2-a/
n0-local-topology-graph-t23-neato.dot.gz
n0-local-topology-tree-t23-dot.dot.gz
n1-local-topology-graph-t23-neato.dot.gz
n1-local-topology-tree-t23-dot.dot.gz
n2-local-topology-graph-t23-neato.dot.gz
n2-local-topology-tree-t23-dot.dot.gz
n3-local-topology-graph-t23-neato.dot.gz
n3-local-topology-tree-t23-dot.dot.gz
n4-local-topology-graph-t23-neato.dot.gz
n4-local-topology-tree-t23-dot.dot.gz
n5-gw2-a-k2-adhoc.log.gz
n5-gw2-a-k2-adhoc.tr
n5-gw2-a-k2-excel.log
n5-gw2-a-k2-stats.log
n5-local-topology-graph-t23-neato.dot.gz
n5-local-topology-tree-t23-dot.dot.gz
n6-local-topology-graph-t23-neato.dot.gz
n6-local-topology-tree-t23-dot.dot.gz
topology-graph-t23-neato.dot.gz
topology-tree-t23-dot.dot.gz
transmission-graph-t23-neato.dot.gz
```

The files prefixed by `nX-local` are node local output files and do not require any global information for their creation, i.e. they are generated only from node local data structures. Files which contain the substring `neato` should be processed by the `neato` tool from the GraphViz suite. The `dot` files should be processed by the `dot` tool. For example, converting node 0s local topology tree into a png output file is done by the following command

```
neato -T png n0-local-topology-graph-t23-neato.dot -o n0-local-
    topology-graph-t23-neato.png
```

assuming the files have been uncompressed before. The topology tree is processed by the command

```
dot -T png n0-local-topology-tree-t23-neato.dot -o n0-local-topology-
    tree-t23-dot.png
```

Of course, we are more interested in the result seen by an omniscient observer. The complete topology tree and topology graph are available in the files `topology-graph-t23-neato.dot` and `topology-tree-t23-dot.dot`. Furthermore, we also export the transmission graph in the file `transmission-graph-t23-neato.dot`. The files should be processed in the same way as the previous examples.

Of special interest are the text files `n5-gw2-a-k2-excel.log` and `n5-gw2-a-k2-stats.log`. These files contain the complete statistical information from the simu-

lation, and we used them to generate all of the following diagrams. An example output is shown in Table 4.1, where the CSV file for Excel has been presented in a tabular format for better illustration purposes.

Table 4.1: Example CVS output from a simulation

Parameter	*Value*
proposals sent	707
nbac sent	313
nbac sent retries	1
nbac acks sent	527
avg. proposal time min.	0.041104
avg. proposal time max.	0.074021
avg. proposal time	0.063804
proposals initiated	180
proposals released	27
full updates sent	21
partial updates sent	0
pwr saving min	3.000000
pwr saving max	14.000000
pwr saving avg	7.368529
pwr saving total	2.827025
pwr required total	3.706364
nbac proposals initiated	27
nbac proposals committed	21
nbac proposals aborted	6
nbac group checks initiated	41
nbac group checks committed	28
nbac group checks aborted	10
convergence time	15.668999
diameter	3
converged	1

We will explain some of the parameters because their meaning is not obvious.

full updates sent: Number of full update messages sent by the link-state service. This is used for neighbor discovery and the exchange of link-state information.

partial updates sent: Always zero in our simulations, because we did not use this feature.

proposals initiated: Proposal searches initiated at a node. In the example above, we have tried 180 times to find a new proposal for a network, but only 27 proposals have been found. The reason for this is that if the topology is nearly complete, it becomes more difficult to find new proposals. If the topology has converged, no new proposals can be found.

proposals released: Proposals released to the TMA. A released proposal creates a NBAC instance and tries to build the new group. As we can see from the data collected for NBAC proposals, 21 of the proposals were built and 6 were aborted.

pwr required total: This is the transmission power required (in dBm) if no TMA is used and every node wants to transmit a single message to its farest neighbor. Any practical analysis will propably want to divide this number by the number of

nodes used. It is estimated by using the information from the link state service which knowns to all its neighbors and the minimum receiver power level. Further information on the calculation of these values is given in Section 4.3.6.

pwr saving total: The total power in dBm which can be saved by using the TMA. Any practical analysis will propably want to divide this number by the number of nodes used.

pwr saving min, pwr saving avg, pwr saving max: Minimum, average and maxmimum possible power saving at all nodes in dB.

convergence time: This is the time when the group structure has changed last at a node.

diameter: The maximum network diameter after the Thallner algorithm has been executed after the algorithm has converged.

converged: Is set to 1 if the network has converged. Is set to 0 if the simulation has been aborted. This is used as an additional check to ensure that no invalid or aborted simulations are used in further analysis.

4.2.2 Convergence detection

Convergence detection was one of the more challenging problems we experienced during our work. The problem is that the algorithm does in fact not stop working after the topology has converged, i.e., it has no built-in convergence detection. Our convergence detection algorithm makes the following assumptions, and we have found that it provides good results in practice and is still easy enough to understand and implement.

- The topology tree is a $k-ary$ tree (allowing some exception at the gateway nodes).
- The average depth of the topology tree is given by the number of groups, that is $\left\lfloor \frac{n-1}{k-1} \right\rfloor$, and the fact that it is a $k-ary$ tree. Therefore we get $avg\_depth = \log_k \left(\left\lfloor \frac{n-1}{k-1} \right\rfloor \right)$.
- To check wheter a group is consistent a group check must be executed. If a node wants to check all its groups it needs the time required for one group check multiplied by the average tree depth. After this time and if no updates have been made to the topology the node can assume that all groups are consistent[2].
- To generate a new proposal, and by assuming a local non-perfect propose module is used, a node must initiate a new search for every group it is a leader in. This proposal must also be accepted by the main algorithm using a NBAC phase and therefore an additional factor has to be added which was simply chosen as 2.

[2]Note that this is not guaranteed because nodes could crash, ... But it is sufficient as a basic convergence criterion, especially if we can assure that no nodes crash within this time window.

- All normal nodes must have a degree of k.

Using all these observations from above we came up with the function presented in Listing 4.4, which calculates the time the network must not have changed to be considered stable. For the impatient reader the calculation is repeated below.

$$
\begin{aligned}
group\_tree\_depth &= \log_k\left(\left\lfloor \frac{n-1}{k-1} \right\rfloor\right) \\
convergence\_time\_prop &= 2 \cdot proposals\_period \cdot group\_tree\_depth \\
convergence\_time\_group &= group\_check\_period \cdot group\_tree\_depth \\
\\
convergence\_time &= \max(convergence\_time\_prop, convergence\_time\_group)
\end{aligned}
$$

Listing 4.4: Calculation of convergence time

```
1  proc convergence-calculate-times { arrname } {
2    global tma topo_thallner_k
3    upvar $arrname times
4
5    set num_nodes [ array size tma ]
6
7    # floor( (n-1)/(k-1) ) = ngroups is the number of groups. If the
8    # tree is balanced the height is log( ngroups ) / log( k ). Every
9    # node therefore walks up this hierarchy and creates group proposals
10   # where 1 is added because of the proposal for the node itself.
11   set group_tree_depth [ expr log( ( $num_nodes - 1 )/( $topo_thallner_k -
       1 ) ) / log( $topo_thallner_k ) + 1 ]
12
13   # add an extra factor of 2 because a generated proposal must also
14   # be accepted by the thallner algorithm.
15   set times(convergence_time_proposals) [ expr [ $tma(0) proposals-period ]
         * $group_tree_depth * 2.0 ]
16
17   # calculate the time required to check all groups at a node
18   set times(convergence_time_group) [ expr [ $tma(0) groupcheck-period ] *
       $group_tree_depth ]
19
20   # the convergence time is the maximum
21   if { $times(convergence_time_group) < $times(convergence_time_proposals)
       } {
22     set times(convergence_time) $times(convergence_time_proposals)
23   } else {
24     set times(convergence_time) $times(convergence_time_group)
25   }
26 }
```

Having such a time available, we simply check that every node did not change its group structure within this time. If this is the case and all normal nodes have a degree of k, then the network is assumed to be stable and the simulation stops. This is implemented in the function 'convergence−test' where the variable is_stable is initially set to 1. If

either a normal node has not degree k or the group structure has changed recently, it is set to 0. The source code is shown in Listing 4.5.

Listing 4.5: Convergence detection in NS2

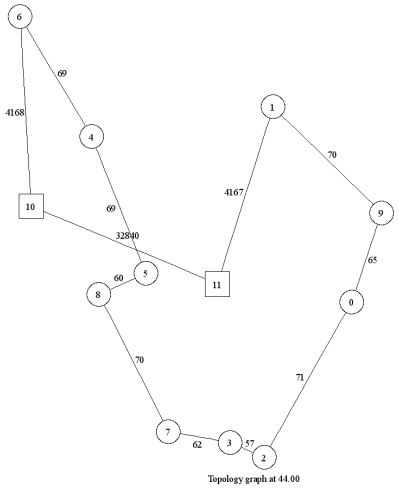
```
1  proc convergence-test {} {
2    global ns tma val last_stats_time topo_prefix topo_thallner_k time_max
3    set scheduler [ $ns set scheduler_ ]
4    set now [ $scheduler now ]
5
6    convergence-calculate-times times
7    stats-convergence-print stdout
8
9    # we assume that the network has converged.
10   set is_stable 1
11
12   # compute the time when the group structures have changed.
13   set last_change_max 0.0
14   for { set i 0 } { $i < [ array size tma ] } { incr i } {
15     set last_change [ $tma($i) last-change ]
16     if { $last_change > $last_change_max } {
17       set last_change_max $last_change
18     }
19     # if this node is not a gateway node it must have a node
20     # degree of k.
21     if { 0 == [ $tma($i) gateway-node ] } {
22       if { $topo_thallner_k != [ $tma($i) node-degree ] } {
23         puts "node [$tma($i) tmaaddr] not gateway node and has node
               degree [$tma($i) node-degree]. not converged!"
24         set is_stable 0
25       }
26     }
27   }
28
29   # if the group structures have not changed during the convergence
30   # time the network is stable.
31   if { $now > $times(convergence_time) } {
32     if { $now - $last_change_max < $times(convergence_time) } {
33       puts "group structure has changed at time $last_change. not converged
             !"
34       set is_stable 0
35     }
36   } else {
37     set is_stable 0
38   }
39   ...
```

If the network is stable or a safety time has been exceeded, we are finished and the results are painted as described in Section 4.3. The safety time has been added to detect bugs in our implementation or to skip very long simulations because of limited computational resources.

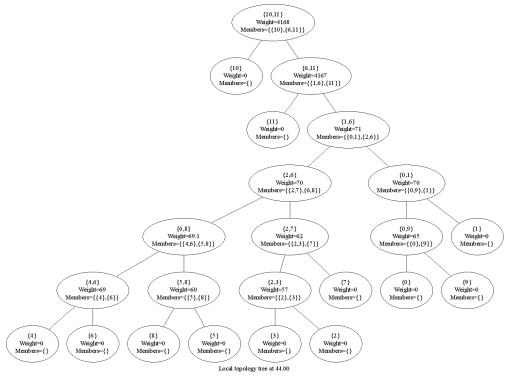
4.3 Results

4.3.1 Example Overlay Graphs

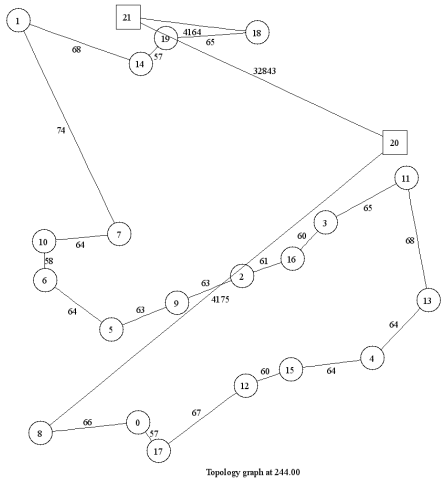
In this section we show some network topology graphs and trees for different network sizes and different values of k. All network topology graphs were created using the local non-perfect propose module with the wireless settings described above. No other traffic was present. Rectangular nodes are gateway nodes and circular nodes are normal nodes. Figures 4.1 and 4.2 show some overlay graphs for $k = 2$. Figures 4.3 and 4.4 show overlay graphs for $k = 3$, and Figures 4.5 and 4.6 for $k = 4$.



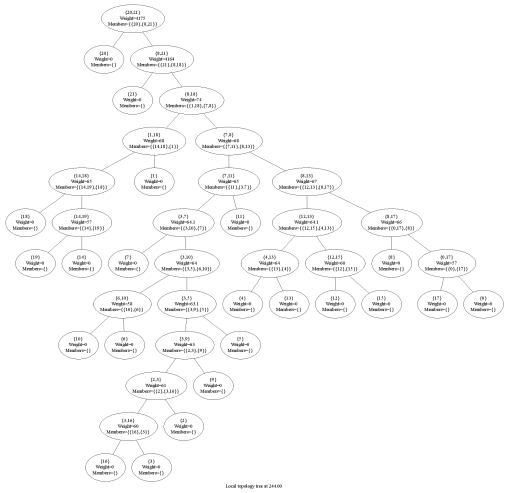
(a) Topology Graph for $n' = 10, n'' = 4$



(b) Topology Tree for $n' = 10, n'' = 4$

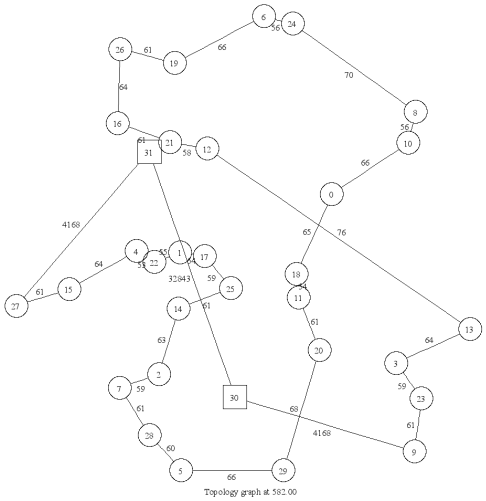


(c) Topology Graph for $n' = 20, n'' = 4$

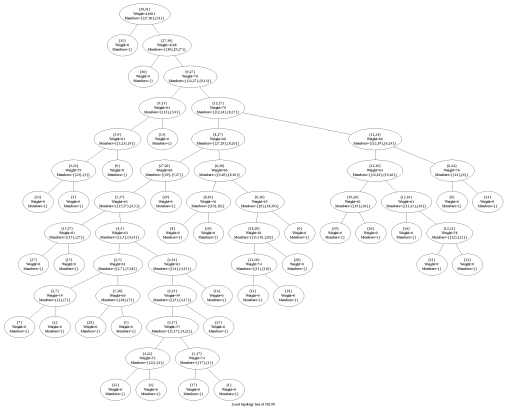


(d) Topology Tree for $n' = 20, n'' = 4$

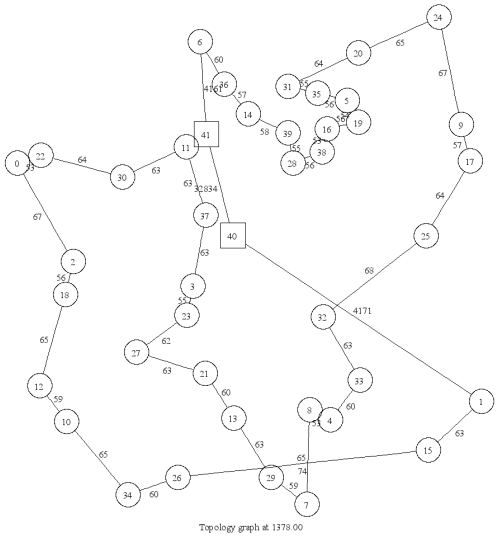
Figure 4.1: Different topology graphs and topology trees for $k = 2$ (part 1/2)



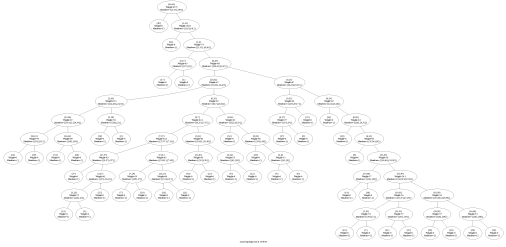
(a) Topology Graph for $n' = 30, n'' = 4$



(b) Topology Tree for $n' = 30, n'' = 4$

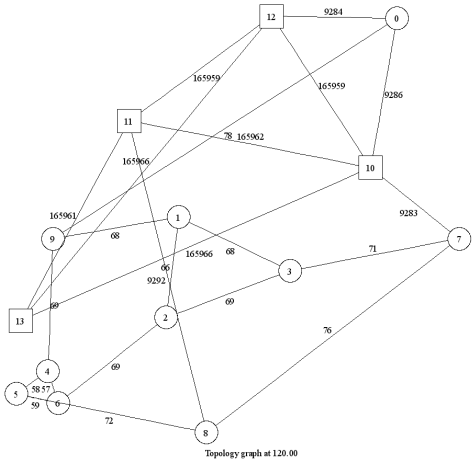


(c) Topology Graph for $n' = 40, n'' = 4$

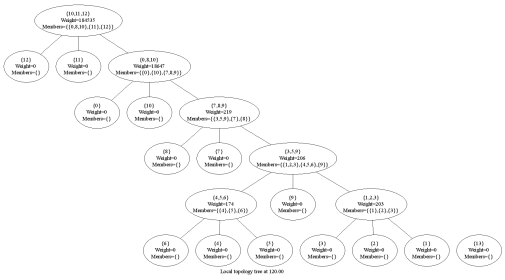


(d) Topology Tree for $n' = 40, n'' = 4$

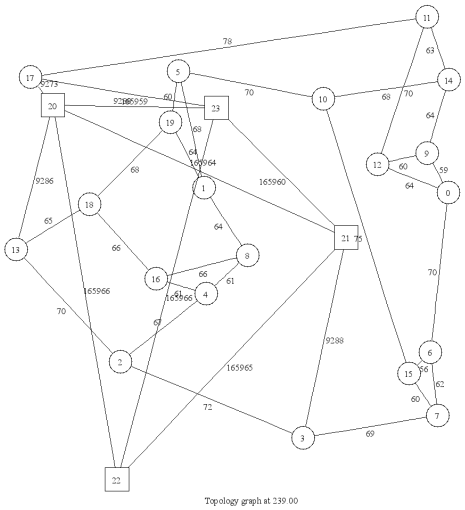
Figure 4.2: Different topology graphs and topology trees for $k = 2$ (part 2/2)



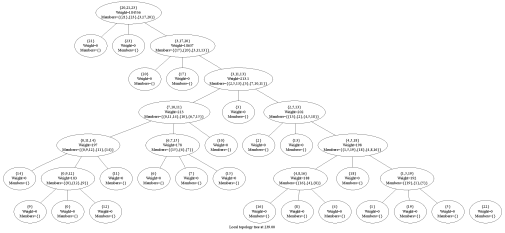
(a) Topology Graph for $n' = 10, n'' = 4$



(b) Topology Tree for $n' = 10, n'' = 4$

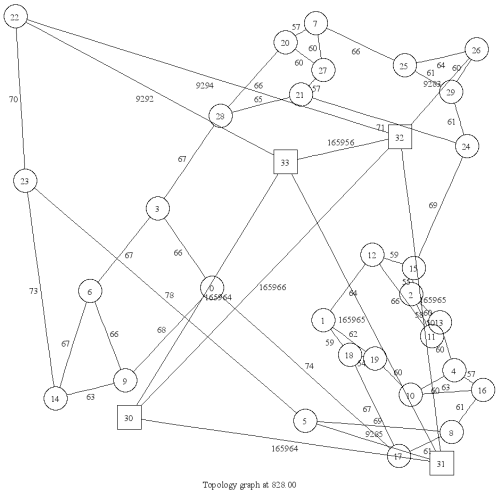


(c) Topology Graph for $n' = 20, n'' = 4$

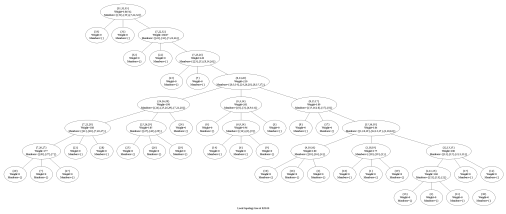


(d) Topology Tree for $n' = 20, n'' = 4$

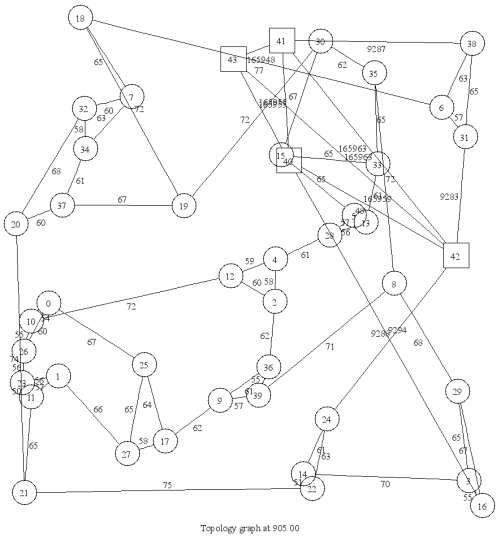
Figure 4.3: Different topology graphs and topology trees for $k = 3$ (part 1/2)



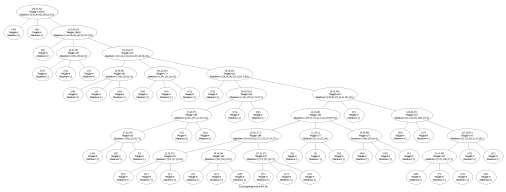
(a) Topology Graph for $n' = 30, n'' = 4$



(b) Topology Tree for $n' = 30, n'' = 4$

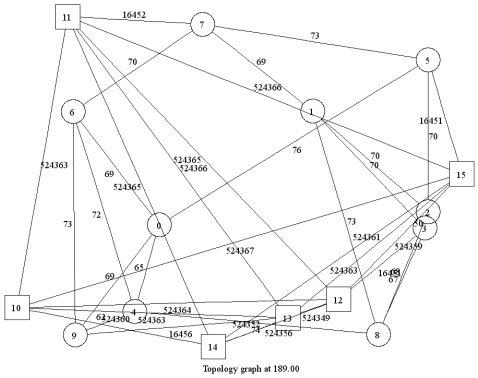


(c) Topology Graph for $n' = 40, n'' = 4$

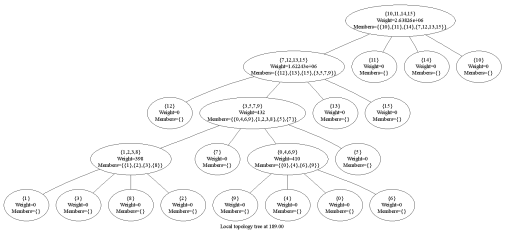


(d) Topology Tree for $n' = 40, n'' = 4$

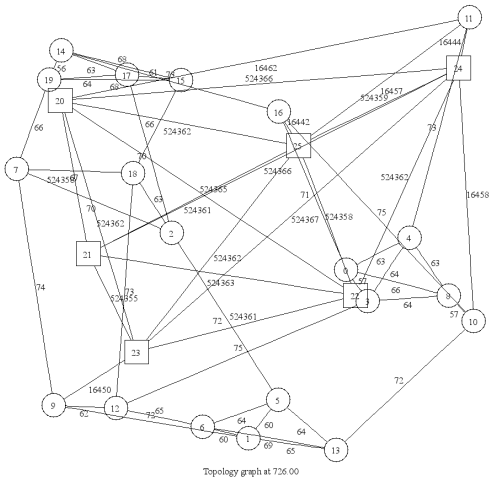
Figure 4.4: Different topology graphs and topology trees for $k = 3$ (part 2/2)



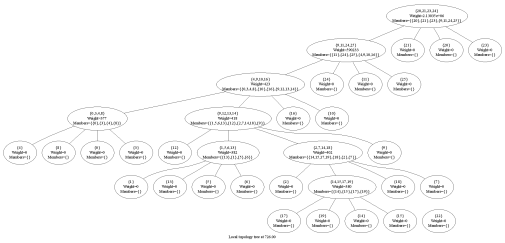
(a) Topology Graph for $n' = 10, n'' = 6$



(b) Topology Tree for $n' = 10, n'' = 6$

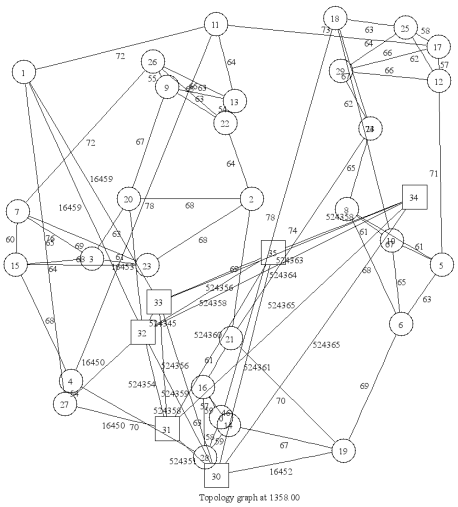


(c) Topology Graph for $n' = 20, n'' = 6$

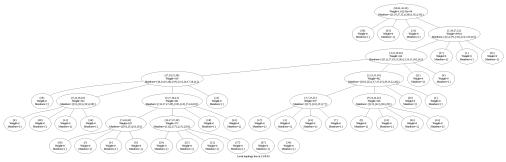


(d) Topology Tree for $n' = 20, n'' = 6$

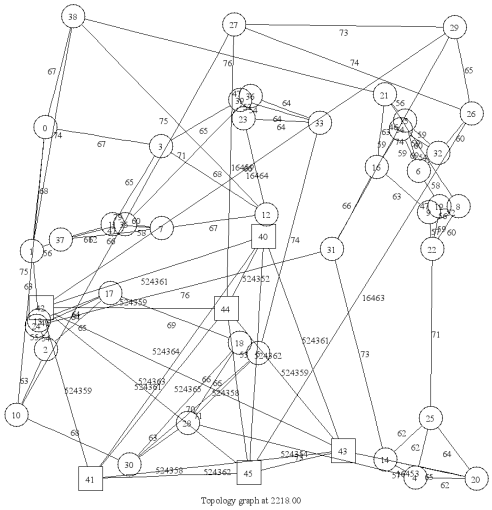
Figure 4.5: Different topology graphs and topology trees for $k = 4$ (part 1/2)



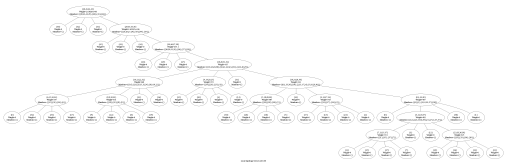
(a) Topology Graph for $n' = 30, n'' = 6$



(b) Topology Tree for $n' = 30, n'' = 6$



(c) Topology Graph for $n' = 40, n'' = 6$



(d) Topology Tree for $n' = 40, n'' = 6$

Figure 4.6: Different topology graphs and topology trees for $k = 4$ (part 2/2)

The next sections will show some results where we focused on message complexity, convergence time and network properties. All results have been generated for 24 variants in increments of 5 number of nodes. That is for $k = 2$ and $k = 3$, we have simulated nearly 500 different network topologies. Because of the computation complexity, we have simulated only 240 topologies for $k = 4$.

4.3.2 Message complexity

Figures 4.7, 4.8 and 4.9 show the relation between the average number of messages sent at a node and the number of nodes in the network for different values of k. The number of nodes in the network n is defined as the sum of n' and n'', where n are the normal nodes and n'' are gateway nodes. The complexity of the link-state service has been excluded and is presented in Section 4.3.7. We can see that the average complexity of $O(n^{2.5})$ [Tha05, p.90] matches with the simulation results of Thallner presented in [Tha05, p.88]. The constants do not match though, but this is caused by the simulation frameworks.

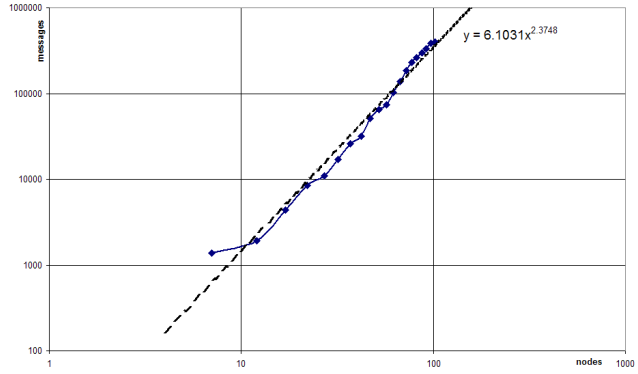


Figure 4.7: Total message complexity for $k = 2$

4.3.3 Convergence time

Figures 4.10, 4.11 and 4.12 show the convergence time for different values of k with respect to the number of nodes. It is important to note that convergence time can be traded for network load. That is, if we decide to increase the load on the network caused by the topology management algorithm, the times scales linearly in a first order approach. So if the network load caused by the TMA is doubled, the time required for convergence will halved. If the load is increased further, one can expect network congestion, and the reliable multicasting service will suffer and ACK explosion would be the consequence. This would actually result in a decreased performance because more messages than necessary would be transmitted because of the retransmission.
We have extrapolated the times with an exponential function where the optimal fit was

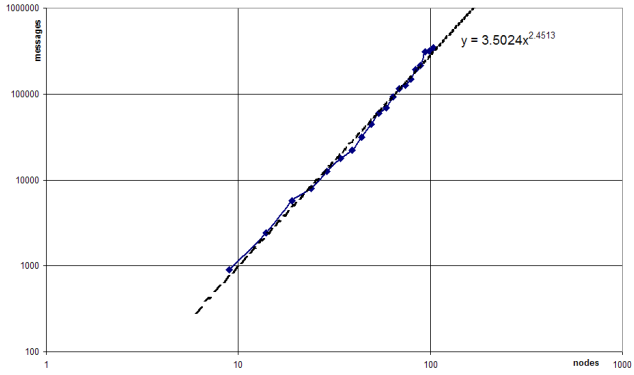


Figure 4.8: Total message complexity for $k = 3$

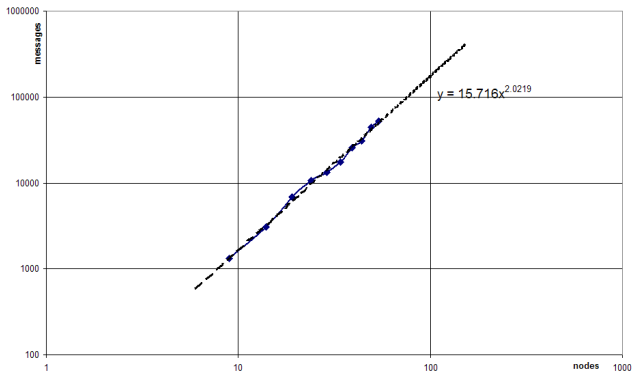


Figure 4.9: Total message complexity for $k = 4$

derived using a least mean square error approach[3]. The simulated average complexity of $\theta(n^{2.7})$ is a bit better than the worst case of $O(n^3)$ for building a topology with a local non-perfect propose module [Tha05, p.82]. It also matches the results from Thallner shown in [Tha05, p.90]. It is interesting to note that the complexity matches because Thallner counted the number of asynchronous rounds where in a round every node receives and sends messages. In our implementation such a round is basically the time window between the generation of proposals.

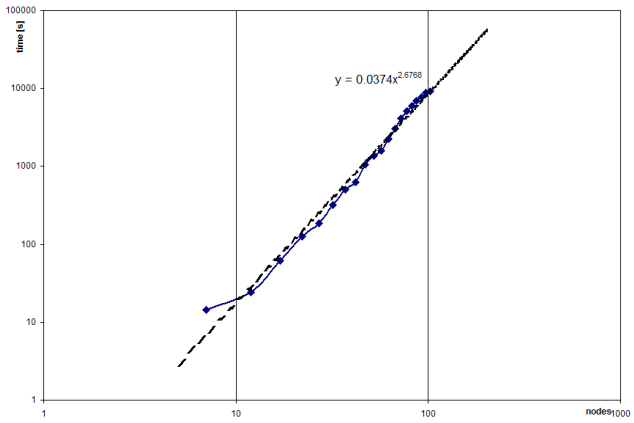


Figure 4.10: Convergence time for different network sizes and $k = 2$

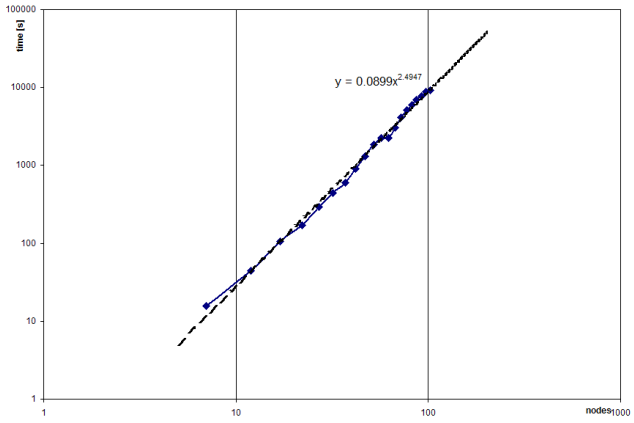


Figure 4.11: Convergence time for different network sizes and $k = 3$

[3] All computations were done using Microsoft Excel.

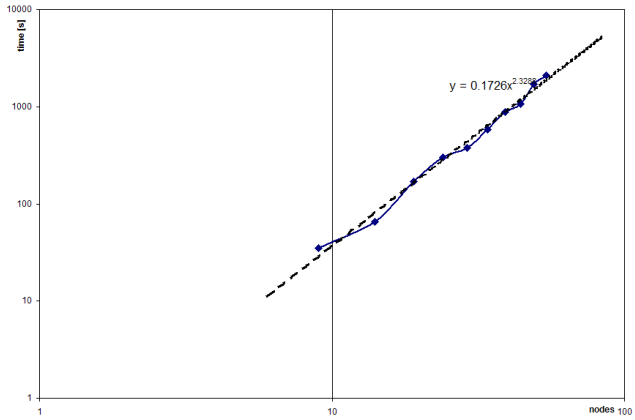


Figure 4.12: Convergence time for different network sizes and $k = 4$

4.3.4 Local-Non Perfect Propose Module

We will now evaluate different performance aspects of the *local non-perfect propose module*. Figures 4.13, 4.14 and 4.15 show the total number of messages sent in relation to the number of nodes for different values of k. Thallner has shown that the worst case complexity is $O(n^{2n})$ in [Tha05, p69.]. It is interesting to see that the average complexity is much better and only shows a linear increase with the number of nodes. Simulation and statistical inaccuracies asides we can observe that the number of messages sent by the local propose module is in $\theta(n)$.

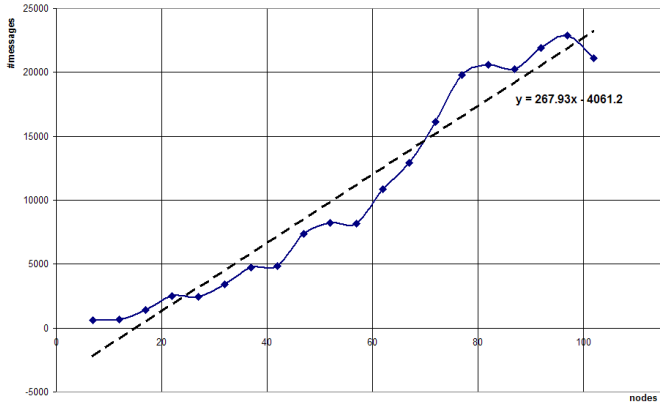


Figure 4.13: Total number of message sent by the LNP propose module for $k = 2$

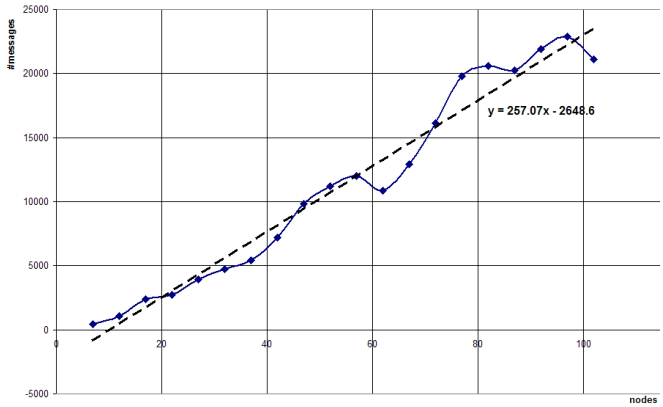


Figure 4.14: Total number of messages sent by the LNP propose module for $k = 3$

Another very interesting metric is the number of messages required for a single search. The *local non-perfect propose* module generates proposals by performing a DFS-search in

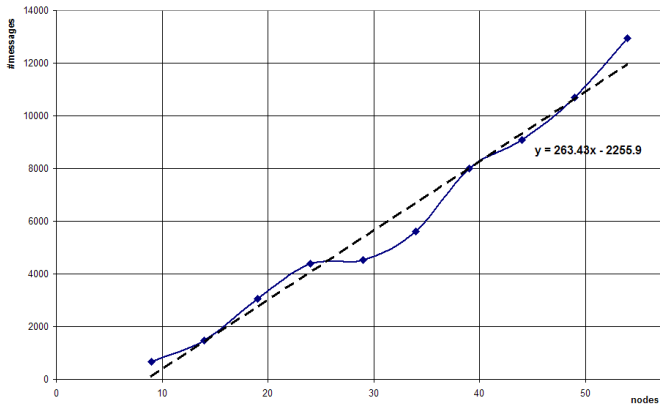


Figure 4.15: Total number of messages sent by the LNP propose module for $k = 4$

the topology tree. Every node is only asked once during the search, and on each node at most $\left\lfloor \frac{n-1}{k-1} \right\rfloor$ groups are queried once [Tha05, p68]. This gives us a worst case complexity of $O(n^2)$ for the number of messages. This is shown in Figures 4.16, 4.17 and 4.18. It is interesting to see that the average complexity seems to be $\theta(\log(n))$. Clearly all results show better performance than the worst case performance of $O(n^2)$ [Tha05, p69.].

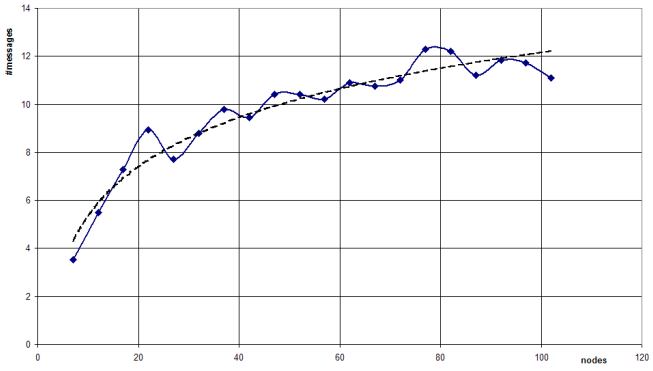


Figure 4.16: Average number of messages for a single proposal for the LNP propose module and $k = 2$

Please note that a single search does not imply that at the end of the search a new proposal is released. This can be seen by looking at the algorithm shown in Section 2.4.3. If the set of potential members, briefly called *pms*, is too small, then the search is not forwarded in line 189. The number of initiated proposal searches and the number of proposals released (=found) are shown in Figures 4.19, 4.20 and 4.21 for different values

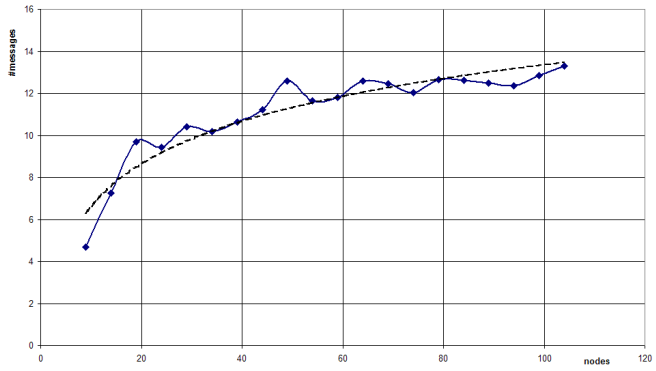


Figure 4.17: Average number of messages for a single proposal for the LNP propose module and $k = 3$

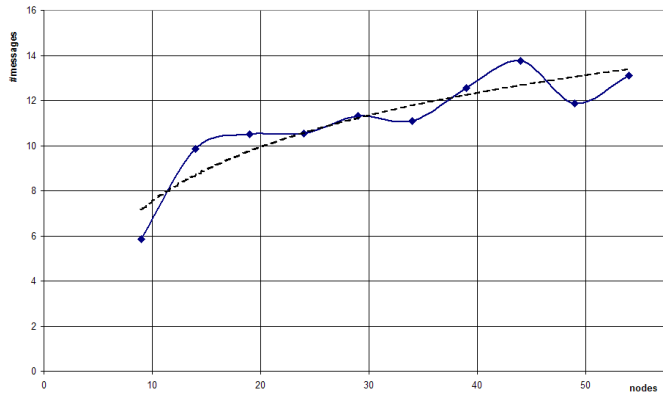


Figure 4.18: Average number of messages for a single proposal for the LNP propose module and $k = 4$

of k.

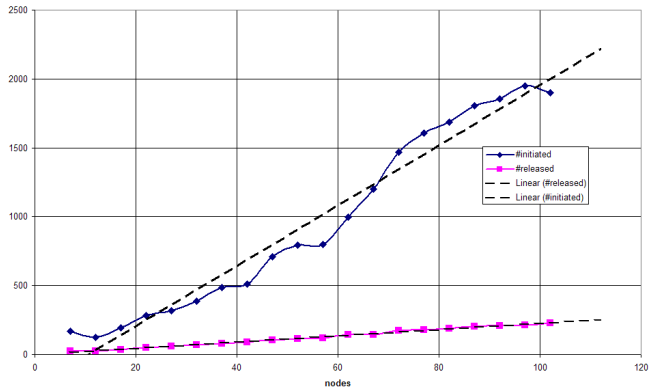


Figure 4.19: Number of proposal searches initiated and actual number of proposals released(=found) for $k = 2$

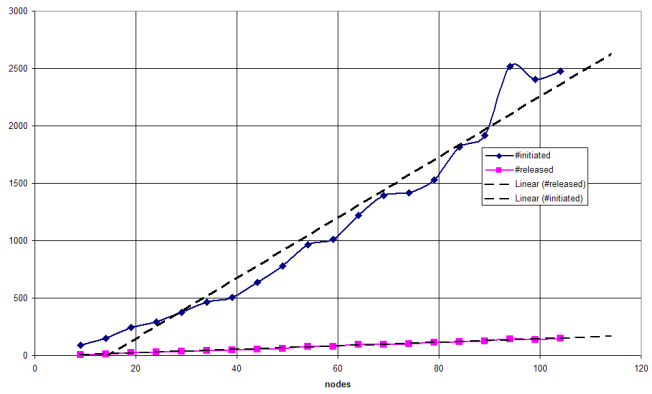


Figure 4.20: Number of proposal searches initiated and actual number of proposals released(=found) for $k = 3$

The release of a proposal to the Thallner algorithm does not imply that this group is eventually built. In Figures 4.22, 4.23 and 4.24 we have plotted the number of released proposals to the NBAC and the number of committed(=built) and aborted(=not built) instances. The reason why some released proposals are aborted is that there is a time difference between the generation of the proposals and the voting on the proposed group. If the topology has changed in the mean time, some nodes might refuse the proposal. Please note that the number of aborted proposals is reasonably low. The reason for this is that the network load was not very high. The longer it takes to find a proposal and to actually build the group, the more likely it is that the group construction is aborted.

We have also evaluated the performance of the local non-perfect propose module with

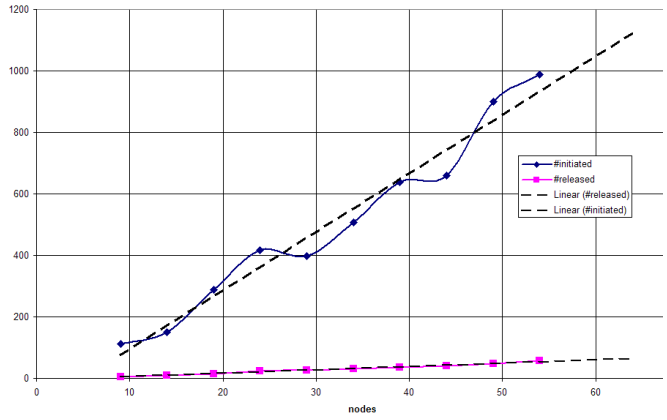


Figure 4.21: Number of proposal searches initiated and actual number of proposals released(=found) for $k = 4$

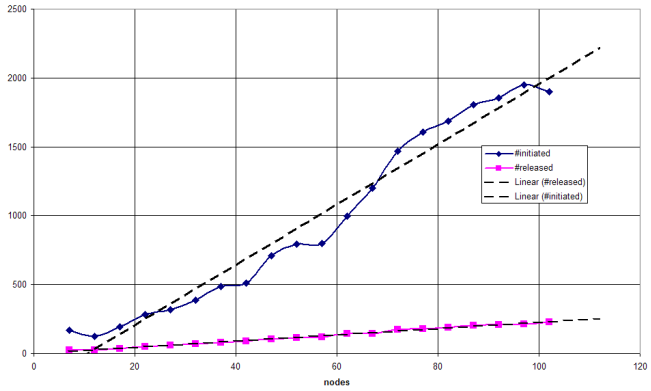


Figure 4.22: Number of NBAC instances for new proposals initiated, committed and aborted for $k = 2$

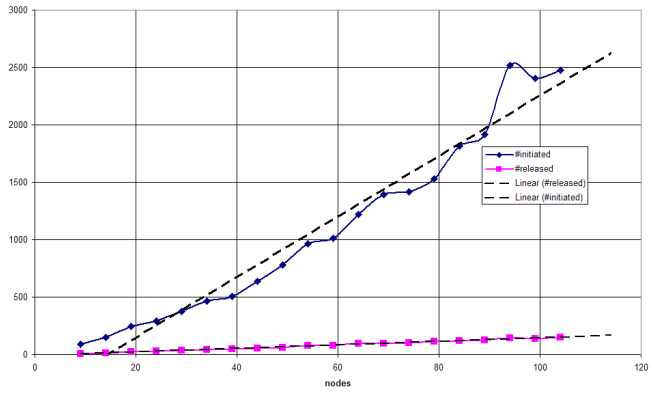


Figure 4.23: Number of NBAC instances for new proposals initiated, committed and aborted for $k = 3$.

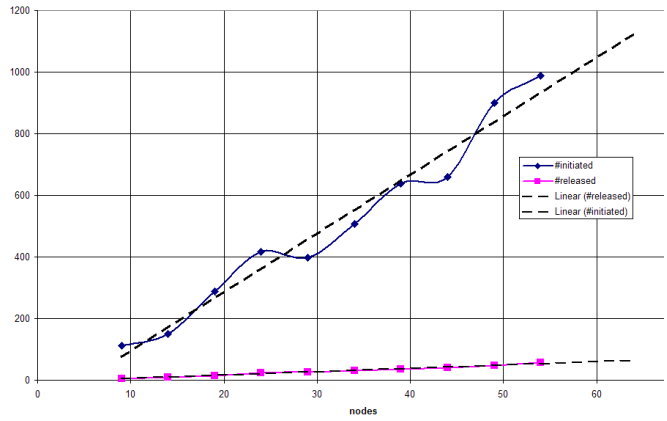


Figure 4.24: Number of NBAC instances for new proposals initiated, committed and aborted for $k = 4$.

respect to the time needed to generate a proposal. This is shown in Figures 4.25, 4.26 and 4.27. It is interesting to see that there is no linear relationship between the average values and the worst case values.

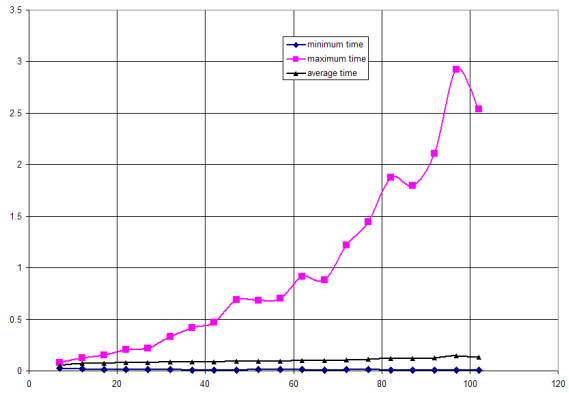


Figure 4.25: Maximum, minimum and average time in seconds required for generating a proposal for $k = 2$

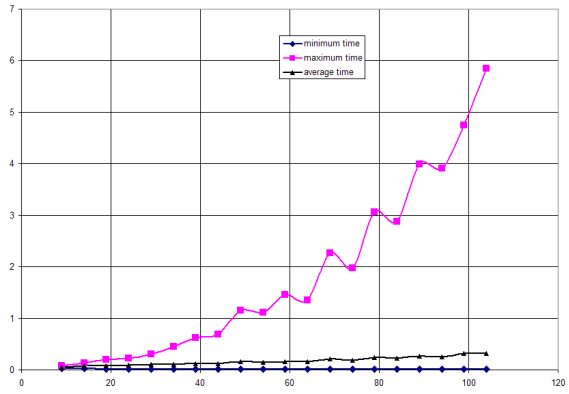


Figure 4.26: Maximum, minimum and average time in seconds required for generating a proposal for $k = 3$.

4.3.5 Group Checking

Group checking is an essential component in the Thallner algorithm and therefore its performance is critical. Again, the group checking components make use of the NBAC service. We have recorded the number of group checks initiated, the number of successful group checks, and the number of aborted groups. Successful in our terminology means that the group was still valid when the check was done. The results are shown in Figures 4.28, 4.29 and 4.30. We can make some interesting observations. First of all it seems

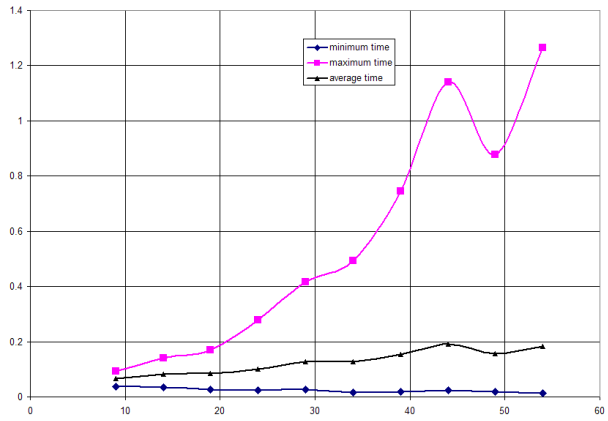


Figure 4.27: Maximum, minimum and average time in seconds required for generating a proposal for $k = 4$.

that the number of group checks required has an average complexity of $\theta(n^3)$. Note that the actual real number strongly depends on the time chosen by our algorithm (of course the complexity remains the same). A very important factor is the time between the generation of proposals and the periodic checking of groups, because a newly generated proposal might invalidate groups. Therefore we have always chosen to do more group checks than we generate proposals.

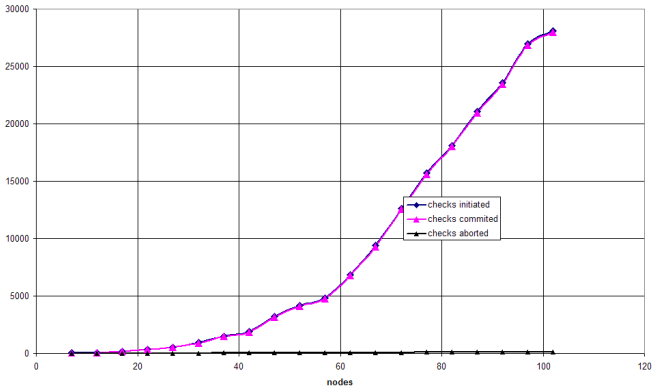


Figure 4.28: Number of group checks initiated, committed, and aborted for $k = 2$

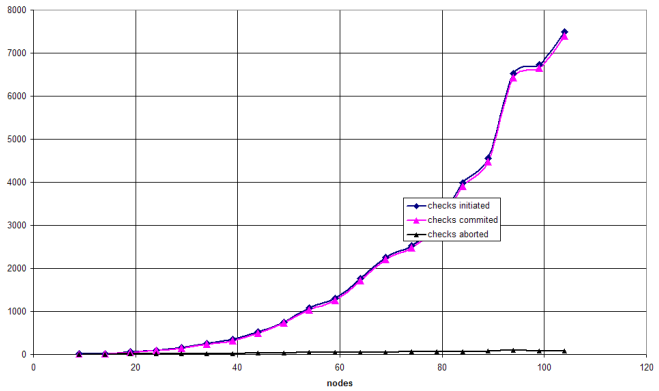


Figure 4.29: Number of group checks initiated, committed, and aborted for $k = 3$

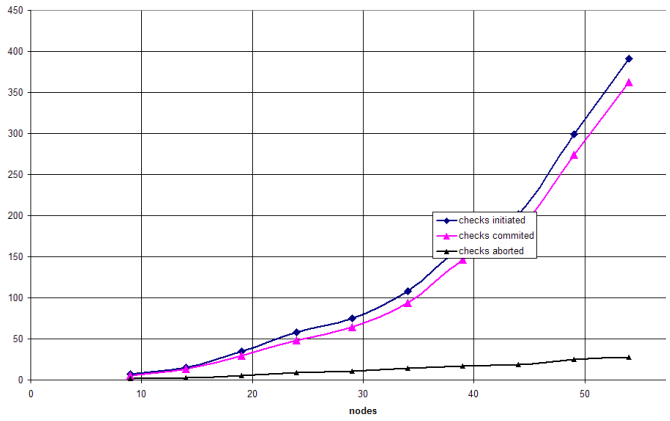


Figure 4.30: Number of group checks initiated, committed, and aborted for $k = 4$

Reliable Multicast performance

The Non-Blocking Atomic Commitment service requires a reliable network transmission. The performance of the reliable multicasting service is therefore critical for the network and we included some statistics from it in our simulation results. Figure 4.31 shows the total number of reliable multicasts sent. Figure 4.32 shows the number of ACKs received. Remember that if a message is sent by the multicast service to a set of nodes, every node must acknowledge this message by an ACK[4]. Therefore the factor of sent multicasts to received ACKs can be used as an estimate for the number of multicast participants[5]. Note that if the network size increases this value will eventually become bigger because top level groups will always have k^2 participants resulting from k members and k terminal nodes for every member. First we assumed that with increasing network size the number would reach k^2 but this is wrong because with a growing network size the number of smaller groups also increases, i.e., groups that have k members but some of these members are single node groups. This is shown in Figure 4.33.

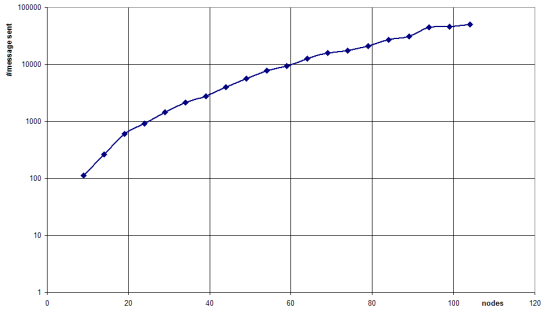


Figure 4.31: Number of reliable multicast messages sent for $k = 3$

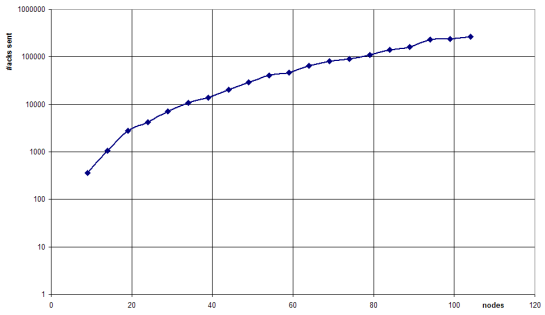


Figure 4.32: Number of ACKs for $k = 3$

[4] Assuming that our ACK based implementation has been used.

[5] Actually we have to calculate the ratio between the number of multicase messages sent and the number of ACKs. Because the originator does not have to send an ACK we finally add 1 to this value.

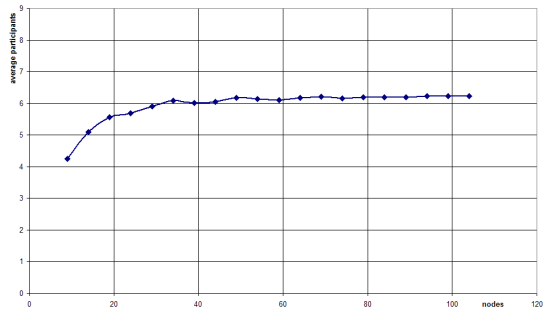


Figure 4.33: Average number of participants for $k = 3$

The following interesting things can be observed. First the message complexity of the reliable multicasting service shows the same order of $O(n^3)$ as the total number of group checks and group proposals initiated by the NBAC service. The reason for this is that an NBAC instance has at most k^2 participants and this value does not depend on the size of the network. Therefore, it is simply a constant factor which does not count in the Landau-Bachmann complexity. The second thing which is quite obvious from the plots is that a reliable multicasting service has a very high performance penalty.

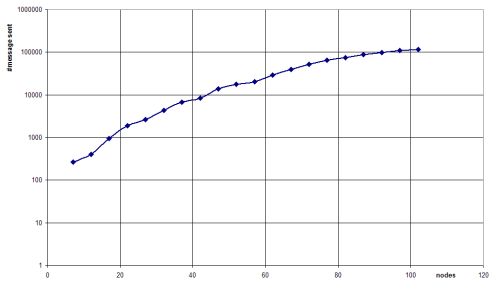
To conclude these simulation results we also present the same figures for $k = 2$ in Figures 4.34(a), 4.34(b) and 4.34(c). The results for $k = 4$ are shown in Figures 4.35(a), 4.35(b) and 4.35(c).

4.3.6 Network properties

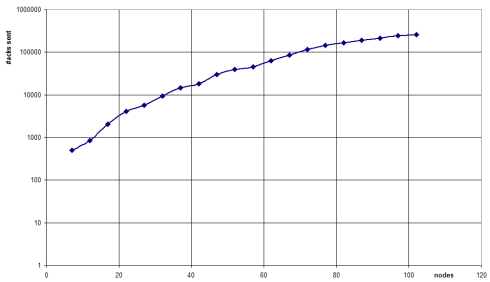
Our algorithm requires a fully connected network for the correct execution of the algorithm and the generation of a topology satisfying our properties. Requiring that every node can talk to each other node requires a large amount of transmission power at every node. We therefore studied if the topology built by the TMA can be used to reduce the transmission power of the network. Our link-state service allows us to calculate the minimum required transmission power at a node, because the link-state messages contain the transmission power, and the physical interface at a node allows us to get the received power level. If we assume that there exists a common lower bound for the correct reception of a message, then we can get the minimum required transmission power by:

$$tx\_pwr\_min[dBm] = \underbrace{(tx\_pwr\_sent[dBm] - rx\_pwr\_received[dBm])}_{pathloss} + rx\_pwr\_min$$

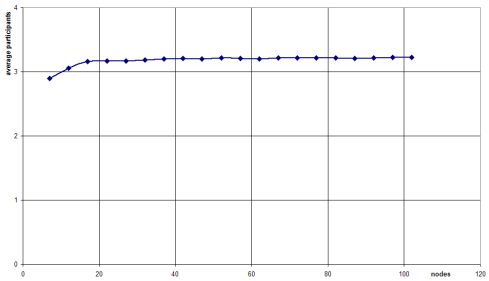
A typical value taken from NS2 for rx_pwr_min is 3.652E-10W which equals $-64.4[dBm]$. This information is stored for every node in the network. The default transmission power is $0.282W$, which equals $24.5[dBm]$. After the network has converged, let



(a) Number of reliable multicast messages.

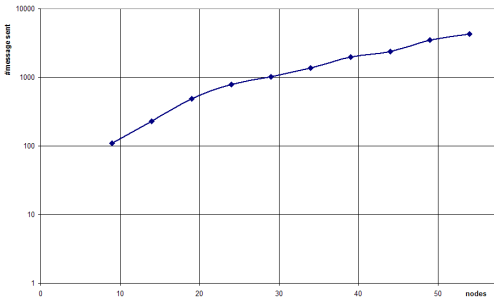


(b) Number of ACKs.

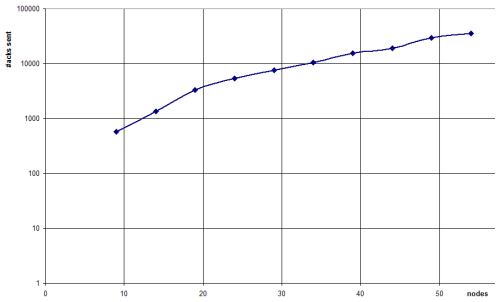


(c) Average number of participants.

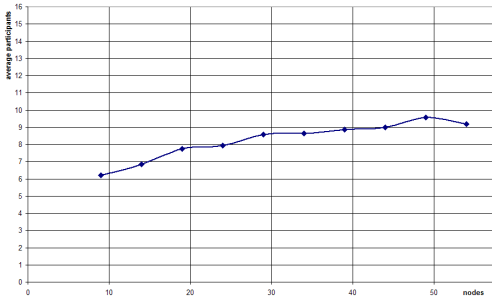
Figure 4.34: NBAC simulation results for $k = 2$



(a) Number of reliable multicast messages.



(b) Number of ACKs.



(c) Average number of participants.

Figure 4.35: NBAC simulation results for $k = 4$

$T = (V, E)$ be the final network topology. We can get the minimum transmission power required at a node i by iterating over its adjacent links.

$$tx\_pwr\_min(i) = \min\left(\{\omega : (i, y, \omega) \in E\}\right) + rx\_pwr\_min$$

The minimum transmission power required when not using the topology management algorithm can be computed by the transmission graph $T' = (V', E')$. In this case we need to talk to every possible neighbor.

$$tx\_pwr\_min'(i) = \min\left(\{\omega : (i, y, \omega) \in E'\}\right) + rx\_pwr\_min$$

We have evaluated these properties for different numbers of nodes and different settings of k. Figures 4.36, 4.37 and 4.38 show the minimum, maximum and average power. The values are calculated as following:

$$\begin{aligned}
pwr_saving_min &= \min_{\forall i \in V}\left(tx_pwr_min'(i) - tx_pwr_min(i)\right) \\
pwr_saving_max &= \max_{\forall i \in V}\left(tx_pwr_min'(i) - tx_pwr_min(i)\right) \\
pwr_saving_avg &= \frac{1}{|V|}\sum_{\forall i \in V}\left(tx_pwr_min'(i) - tx_pwr_min(i)\right)
\end{aligned}$$

Note that we have excluded gateway nodes from this calculation, because they are fully connected and their connections are not touched by the TMA. We can see that the power saving increases with the number of nodes. The reason is quite simple: Because the grid used to place the nodes remains the same, a decently performing propose module will always build groups with a low average weight. This implies that nodes are nearby, and therefore they do not require great amounts of transmission power.

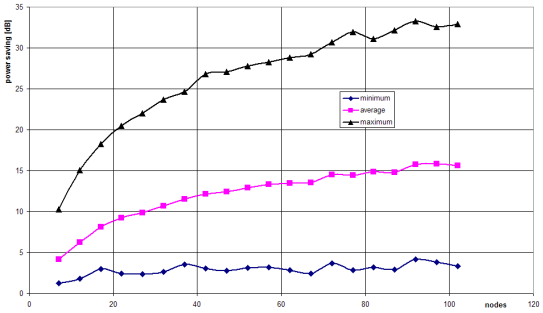


Figure 4.36: Minimum, maximum and average power saving for $k = 2$

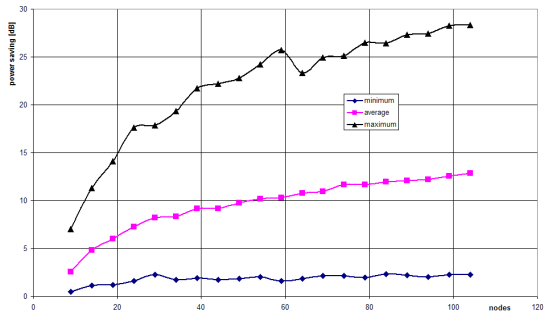


Figure 4.37: Minimum, maximum and average power saving for $k = 3$

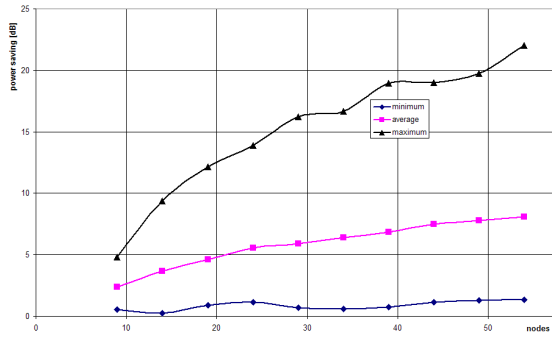


Figure 4.38: Minimum, maximum and average power saving for $k = 4$

4.3.7 Other results

Another component in our simulation network is the *link-state service*. This service is used extensively by the propose modules, because they need to obtain connectivity information. Our basic link-state service is an efficient implementation in a broadcast network and has a message complexity of $O(n \cdot t)$, where t is the simulation time and n is the number of nodes. The message complexity is shown in Figure 4.39. In the author's opinion this is the best one can achieve. The reason for this is that the aliveness of a node can only be assured by receiving a message from that node. Therefore , every node needs to send periodic messages, which implies a factor of n because there are n nodes. Furthermore , we need periodic information, and therefore we have an additional factor of t. Our claim is supported by plotting the factor of the number of messages divided by the product of nt, which is shown in Figure 4.40. Note that the exponential graph in Figure 4.39 comes from the exponential time required for network convergence. Or in other words: Because the link state service produces a constant number of messages within a time window, but the convergence time increases exponentially with the number of nodes, the number of messages for the link state service also increase exponentially with the number of nodes.

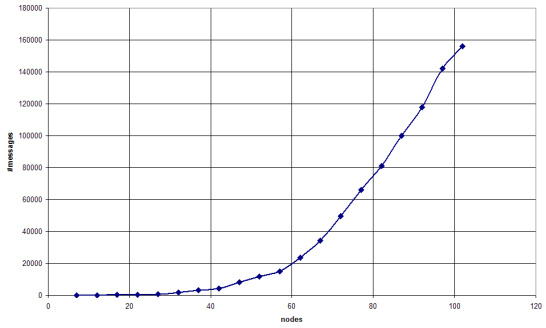


Figure 4.39: Number of linkstate messages

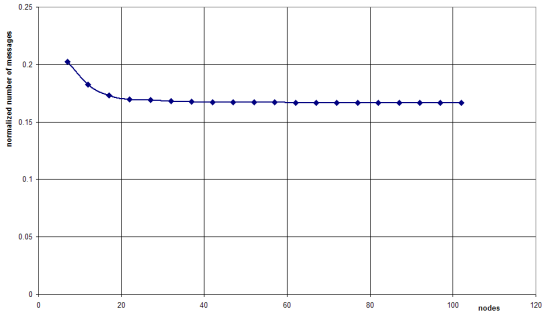


Figure 4.40: Number of linkstate messages normalized to $n \cdot t$ product

Another interesting property of a (connected) graph is its diameter. Various diameters for converged topologies are shown in Figures 4.41, 4.42, and 4.43 for different values of k. Note that for $k = 2$, the diameter is always $n/2$ because the TMA establishes a 2-connected graph, which is essentially a Hamiltonian cycle.

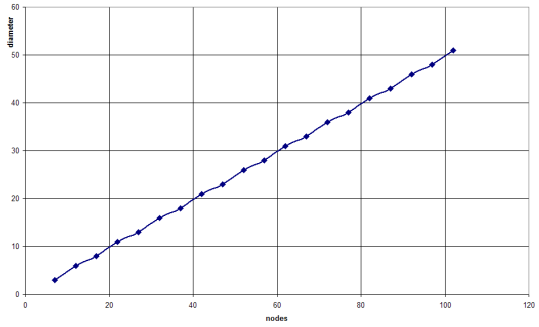


Figure 4.41: Network diameter for $k = 2$

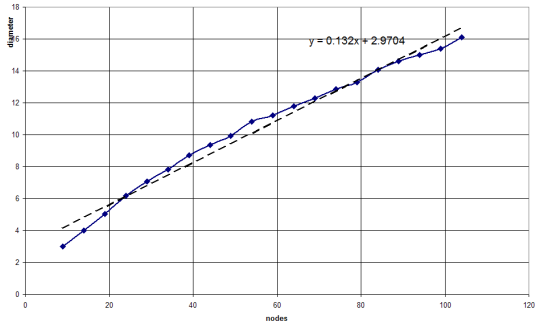


Figure 4.42: Network diameter for $k = 3$

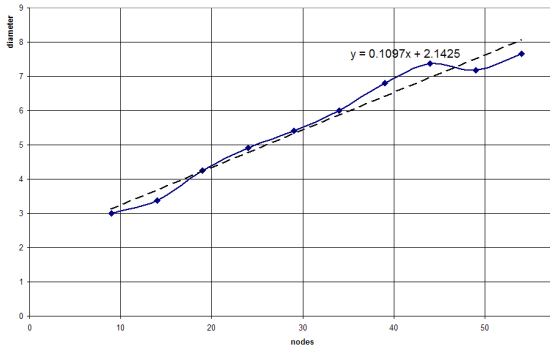


Figure 4.43: Network diameter for $k = 4$

5 The "Extended-" Thallner Algorithm

5.1 Introduction

During our studies, we noticed that the complexity of the *group checking* and *group proposals*, which require the use of a non-blocking atomic commitment protocol, is very high. The complexity is about $n+2 \cdot n^2$ for an unreliable network with no node failures when a reliable multicast service using an ACK based protocol is used. The first term is for the initial message together with the required ACKs and the last term is for the votes and the finalization. In a network where node failures can happen the message complexity increases even more because a consensus algorithm is needed. The actual performance of these algorithms can be improved, if some of the networking assumptions are weakened. Therefore we decided to think about some possible alternate implementations of these components, and we would like to show our results in this chapter.

We started with the classic paper from M. Raynal [Ray96] which presents approaches for solving the non-blocking atomic commitment problem in synchronous and asynchronous networks. Because our primary application target are wireless ad hoc networks, we looked at typical properties of wireless ad hoc networks and came up with the following list.

- If a wireless frame is transmitted over a wireless medium and there are no intermediate nodes, i.e., no additional routing or forwarding of messages is performed, then the transmission time can be easily bounded if some additional assumptions are taken. Basically, one has to ensure that these packets are not queued at either the sender- or receiver interface and are processed as fast as possible. Using appropriate scheduling techniques real-time processing within a bounded time is possible.

- If such a bounded time for transmission can be established, wireless messages are either received within a given time frame δ or not at all, i.e., they are lost. Furthermore, the fact that a message has been lost can be detected by the receiver, assuming that the receiver knows that a message was sent.

- A wireless medium is a broadcast channel and therefore all algorithms should make use of this fact to reduce message complexity. This is particularily important for tasks like consensus, the exchange of votes, ... where we have a set of nodes and all nodes need to know the same information from a single node.

These properties are mentioned here only as an introduction, and a more formal list is provided later in this chapter. However, looking at these properties we can see that this allows the implementation of a synchronous system model with unreliable links. Therefore we have chosen a basic NBAC protocol for a synchronous network, which is similar to the one proposed by Raynal in [Ray96, p.12]. The differences are that the Thallner algorithm requires the use of an NBAC protocol with the modifications that we can piggy-back additional information with the votes, run multiple instances in parallel, and we have finalization phase[1]. The algorithm is shown in Listing 5.1.

[1]Finalization deals with the problem of knowing that all nodes have executed their decision.

Some of the data types used are not obvious and therefore require an additional explanation: The mapping vote is used to collect the votes from all participants for a given NBAC UID (=unique identifier). A NBAC UID is a unique global id which is implemented by a local sequence counter and the unique MAC address of a node. The functions 'vote_IMPL' and 'decision_IMPL' are instances of an application dependent decision and voting function[2].

Listing 5.1: A NBAC protocol for synchronous systems

```
1  function initiate( participants, data )
2  begin
3    // used to run multiple instances of the NBAC protocol at once.
4    var nbac_uid ← global_unique_id( )
5    // send multicast with the data to vote upon.
6    multicast_send( participants, (nbac_uid, data) )
7  end
8
9  function multicast_deliver( source, participants, data )
10 begin
11   // extract the data from the multicast message payload
12   var (nbac_uid, nbac_data) ← data
13   nbac(nbac_uid, participants, nbac_data )
14 end
15
16 function s_rel_multicast_deliver( source, participants, data )
17 begin
18  // extract the data from the multicast message payload
19   var (nbac_uid, vote, commit_data) ← data
20   vote(nbac_uid, source) ← (vote, commit_data)
21 end
22
23 function nbac( nbac_uid, participants, data )
24 begin
25   // let node vote on the data using the application dependent
26   // function vote_IMPL.
27   var (vote, commit_data) ← vote_IMPL(nbac_uid, data)
28   // synchronous reliable multicast requires time δ'
29   s_rel_multicast_send( participants, (nbac_uid, vote, commit_data) )
30   set timer = create_timer(δ + δ')
31   wait for
32     // all votes have been received
33     ∀v ∈ participants : vote(nbac_uid, v) = COMMIT
34     // one participant wants to abort
35     ∨ ∃v ∈ participants : vote(nbac_uid, v) = ABORT
36     // timer has expired
37     ∨ expired(timer)
38
39   if ∃v ∈ participants : vote(nbac_uid, v) = ABORT then
```

[2] In the Thallner algorithm these are the appropriate functions for group checking and group proposals.

```
40      var result ← ABORT
41    if expired(timer) then
42      var result ← ABORT
43    if ∀v ∈ participants : vote(nbac_uid, v) = COMMIT then
44      var result ← COMMIT
45
46    var global_commit_data ←
          {commit_data : ∀v ∈ participants(∃(vote, commit_data) ∈ vote(nbac_uid, v))}
47    decision_IMPL(nbac_uid, result, data, global_commit_data)
48  end
```

It has been shown by Raynal in [Ray96, p.11-p.13] that the algorithm in Listing 5.1 solves the NBAC problem if all networking primitives used are available. The properties and the definition of the NBAC problem, as well as the basic networking primitives, are repeated below for presentation purposes and are based on [Ray96].

A NBAC protocol assures that all participants take the same decision, which is either **COMMIT** or **ABORT**. The exact meaning is application dependent, but committing typically means making changes permanent and abort cancels all work. Whether the outcome is **COMMIT** or **ABORT** depends on the votes from the individual participants and on network or node failures.

Definition 32 (Non-blocking atomic commitment - NBAC)**.** *A protocol solving the NBAC problem satisfies the following properties:*

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 **COMMIT**.*

Non-Triviality: *If all participants vote **COMMIT** and there is no failure the outcome decision is **COMMIT**.*

Let $P = \{p_1, \ldots, p_n\}$ be a set of participants. The most basic primitive is 'multicast_send', which sends a message containing some information to a set of participants. If multicast addresses are available for every possible group, an implementation would simply send the message to this address. In a broadcast network an efficient implementation is to send the message to the broadcast address and include the set of participants in the data payload. This data payload is then inspected by the receiving nodes and if the message is targeted for that node it is delivered at that node. An obvious drawback of this primitive is that it is not fault-tolerant because it cannot be guaranteed that all nodes of P will actually deliver the message. A very powerful primitive is the ' s_rel_multicast_send ' which reliably sends a message m to a set of process. It has been defined by Raynal in [Ray96, p.9] as:

Definition 33 (Reliable multicast in synchronous systems)**.** *The aim of the primitive s_rel_multicast_send (P, m) is to reliably send a message m to all participants P with an "all-or-none atomicity" property.*

Termination: *If a correct process p multicasts a message m to the set of participants P, then some correct process delivers m (or all processes are faulty).*

Validity: *If a process p delivers a message m, then m has been multicast to a set of participants P and p belongs to this set.*

Integrity: *A process p delivers a message m at most once.*

Uniform agreement: *If any process of P delivers m, then all correct processes deliver m.*

Timeliness: *There is a time constant δ' such that if the multicast is initiated at time t, no process delivers a message m after $t + \delta'$.*

The big problem is that it is impossible to implement the synchronous reliable multicast primitive in a network with unreliable links.

Theorem 2. *Let $T = (V, E)$ be a transmission graph which is assumed to be fully connected initially and let $P = \{p_1, \ldots, p_n\}$ be a set of participants for a synchronous reliable multicast and let $n > 2$. Then it is impossible to implement the primitive ' s_rel_multicast_send ' in a synchronous network with an unbounded number of link failures.*

Proof. Let p_i be any node in P which is not the initiator of the multicast. If all links in E fail where p_i is an incident node for that edge, then p_i is isolated. Still the process is correct and therefore the uniform agreement is violated. □

Knowing this fact we have chosen to weaken the network model. This is presented in the next section, where we also provide some justification why we think that this network model actually matches the real world environment.

5.2 Network model

We would like to present our *time-bounded unreliable transmission* network model, briefly TBUT network model, which is given in Definition 34.

Definition 34 (TBUT network model)**.** *The* time-bounded unreliable transmission *(TBUT) network model makes the following assumptions:*

1. *We assume a synchronous system where all correct transmission and computation times are bounded. If a message m is sent by a node and is received correctly, and the receiving node is correct, the message is processed within a known, finite and constant time δ. A message is considered as processed when it has been passed to the application layer where it can be evaluated by an algorithm.*

2. *Unicast message transmission is unreliable. That is, if a message m is transmitted at time t by a node p_i to a node p_j, it is either processed by p_j at the time t' with $t < t' \leq t + \delta$ or not at all. If a message is not received by a node it is counted as a link failure.*

3. *Broadcast message transmission is unreliable. If a message m is transmitted at time t by a node p to the broadcast address with receiver set V, then each node $p' \in V$ either receives m at the time t' with $t < t' \leq t + \delta$ or not at all. If a message is not received by a node it is counted as a link failure.*

4. *We assume that within a given (and hopefully short) time frame Δ at most $f \leq n-2$ nodes **and** links fail in total. In our case, this time frame Δ encloses the complete modified NBAC transaction. It is therefore important to keep this time as short as possible, because limiting the number of faults within a given time does reduce the mobility[3].*

5. *Messages cannot be corrupted.*

6. *Is a message is receives by a process p. it has been sent by some process.*

7. *Nodes are fail-silent[4]. Late joining or rejoining of nodes is allowed, but not within an open transaction. That is, if a transaction spanning a time Δ is in progress and node x has failed and participated in this transaction, it is not allowed to rejoin before the time window Δ has expired.*

8. *The network does not need to be fully connected. That is, for a transmission graph $T = (V, E)$, there can exist nodes $x, y \in V$ where there exists no edge $(x, y) \in E$. Note that for participants in a modified NBAC, a missing link is counted as a failure.*

9. *We assume that every node has access to a local clock for the implementation of a timer. Clocks need to satisfy a global drift condition, but we do not require them to be synchronized. Let $C_1(t)$ be the value of the local clock of node p_1 at time t, and $C_2(t)$ be the value of p_2's local clock at time t. Let $t' = t + \delta$. Then we require that $||C_1(t') - C_1(t)| - |C_2(t') - C_2(t)|| < J$ with $J \in \mathbb{R}^+$.*

The assumptions (1), (2) and (3) are reasonable in a wireless network. If a message is transmitted, the propagation time is only limited by the wave propagation time of the wireless media (=speed of light). Therefore, if a message is sent by a transmitter, it is either received within a given time frame or not. Please note that this must include any local and remote queuing delays. Assuming that a node has sufficient computational power, it can process the message within a finite time. Adding these two times defines our time δ. For (3), it is important to note that there can exist nodes which receive

[3] This comes from the fact that a node might move out of the transmission range, which is treated as a transmission error, or that the communication becomes unreliable.

[4] For example if their energy is exhausted.

the message and nodes which do not. Therefore message transmission can simply be implemented by sending to the MAC broadcast address. Requirement (4) is covered in more detail later, and is a result of our modified algorithm. It simply limits the number of tolerable failures within a given time frame. (5) is implemented by using CRC checksums for the frames. Problem number (6) is dealt with by adding a startup timeout. Assumption number (8) might seem difficult but it should be noted that the clocks need not be synchronized, which is the difficult part. In fact the implementation is quite easy if a high speed timer is available and the timer ticks are used to implement so called macro ticks. For presentation purposes we assume initially that $J = 0$ and will only take it into account in our final algorithm.

In the following section, we will present the implementation of the algorithm in our system model.

5.3 Implementation

We will start by showing how the reliable multicast primitive can be implemented in our modified network model and will then analyze the complexity of the complete algorithm, which was the primary reason for seeking an alternate approach. In the second part of this section, we will present an alternate solution which solves the same problem using a different approach but with a better message complexity.

5.3.1 Implementation using the Synchronous Reliable Multicast

We have already proved in Theorem 2 that in general it is not possible to implement a reliable multicast if the links are unreliable and an unbounded number of links can fail. But this situation is different in our modified network model. We start by claiming the following Lemma.

Lemma 1. *Let $P = \{p_1, \ldots, p_n\}$ be a set of participants which are fully connected and assume our system model from Definition 34. W.l.o.g assume that process p_1 wants to multicast a message to all participants. If every node broadcasts the message again before locally delivering it we can guarantee for $f \leq n - 2$ link and node failures in total, that either all correct nodes deliver the multicast message m within two rounds (equaling 2δ), or it is not delivered at all.*

Proof. The first case is that no participants delivers the message. Let us assume that p_1 crashes before multicasting the message to all participants. Then no participant, including p_1, delivers the message. The reason for this is that p_1 by itself does only deliver the message locally after it has multicasted it to all participants. Furthermore the message has never been sent over the multicast channel and therefore no other node can receive it.

Now assume that p_1 manages to broadcast its value. In that case, the receiver set gets parted into two sets, P_1 respectively P_2 of nodes that did, respectively did not receive the message from p_1. Clearly $|P_2| = f' \leq f$, and $|P_1| \geq n - 1 - f'$. In the second round,

all correct processes in P_1 attempt to forward the message from p_1. As there are only $e \leq f - f'$ errors left, with $f \leq n - 2$, at least $n - 1 - f' - e \geq n - 1 - f \geq 1$ processes in P_1 will not be hit by faults in the second round and thus forward p_1's message to all processes in P_2.

□

Listing 5.2 shows an algorithm which solves this problem in our modified network model and we claim the following properties:

Theorem 3. *The algorithm shown in Listing 5.2 implements the reliable multicast primitive defined in Definition 33. A message can be sent by invoking ' s_rel_multicast_send ' and any message delivered is passed to the application by the invocation of ' s_rel_multicast_deliver '*

Proof. Let $P = \{p_1, \ldots, p_n\}$ be a set of multicast participants and let us assume w.l.o.g that p_1 is the process which has initiated the reliable multicast by invoking the primitive ' s_rel_multicast_send '. We will first deal with the special case when p_1 crashes before executing line 23. Then no process receives the message and all properties hold trivially. Now let us assume that p_1 executes line 23. The algorithm is essentially an implementation of the algorithm described in Lemma 1 and therefore we have the *Uniform Agreement* property. *Validity* is obvious from the algorithm because the set of participants is included in the message and it is checked in Line 48 and only values sent from participants belonging to this set are taken into account. *Termination* is also simple. If the process is correct it can send the message in line 23. If there exists another correct process it receives m (see the beginning of this proof). Because it has not received the message before, the test in line 54 will pass and it will deliver the message giving us the desired result. *Integrity* is also implemented by the check in Line 54. If a message has been received its sequence number is added to this set. Sequence numbers are only removed after 2δ which is essentially the time required for our broadcasting algorithm as shown in Lemma 1. Therefore it is safe to remove the old sequence numbers afterwards[5]. □

Theorem 4. *In our network model, the message complexity of the algorithm shown in Listing 5.2 is in* $\Theta(n)$.

Proof. There are n participants. At the beginning the initiator sends the message, which accounts for 1 message. At most $n - 1$ participants can receive the message. Every participant only forwards the message once, and therefore we have n messages in total. Therefore the message complexity is constant giving us the desired result. □

Theorem 5. *There exists a time constant* $\delta' = 2\delta$ *for the algorithm shown in Listing* 5.2 *such that if the multicast is initiated at time* t*, then no process delivers the multicast after the time* $t + \delta'$.

[5]Note that this is not required for the correctness, but any practical algorithm requires this because otherwise the memory would grow unbounded.

Proof. If the initiator of the multicast sends a message at time t, then all processes receive this message at most at $t + \delta$ by our network model assumption. If the local processing takes no time, which we assume here, then every correct process which has received the message broadcasts it again. Again these messages take at most δ. Furthermore, a process only forwards a message once, which gives our desired result. □

Listing 5.2: Simple- and reliable multicast networking primitives

```
1  var ID // network wide node unique ID.
2  var local_seq_counter ← 0
3  var remote_counters[v] ← ∅ ∀v ∈ V
4
5  function multicast_send( participants, data )
6  begin
7    // invoke low level primitive to broadcast message on network.
8    lowlevel_multicast( MSG_TYPE_SMCAST, participants, data )
9  end
10
11 function multicast_deliver( source, participants, data )
12 begin
13   // application dependent delivery function for basic
14   // multicast.
15   ...
16 end
17
18 function s_rel_multicast_send( participants, data )
19 begin
20   local_seq_counter ← local_seq_counter + 1
21   var mcast_uid ← (ID, local_seq_counter)
22   // invoke low level primitive to broadcast message on network.
23   lowlevel_multicast( MSG_TYPE_RMCAST, participants, (mcast_uid, data) )
24 end
25
26 function s_rel_multicast_deliver( source, participants, data )
27 begin
28   // application dependent delivery function for reliable
29   // multicast.
30   ...
31 end
32
33 // low level multicast primitive.
34 function lowlevel_multicast( msg_type, participants, data )
35 begin
36   // implement multicast by sending to the broadcast channel and
37   // by including the actual participants in the payload.
38   broadcast( msg_type, (participants, data) )
39 end
40
41 // any message
```

```
42 function lowlevel_receive( source, msg_type, payload )
43 begin
44   // reliable multicast message.
45   if msg_type = MSG_TYPE_RMCAST
46     var (participants, data) ← payload
47     // check if this message is for us.
48     if ID ∈ participants
49       // unwrap the internal data payload.
50       (mcast_uid, data2) ← data
51       (remote_ID, remote_seq_counter) ← mcast_uid
52       // test if we have already received this message
53       sequence_numbers = {seq : (seq, timestamp) ∈ remote_counters[remote_ID]}
54       if remote_seq_counter ∉ sequence_numbers
55         // multicast it again before delivering it locally.
56         lowlevel_multicast( MSG_TYPE_RMCAST, participants, data )
57         s_rel_multicast_deliver( remote_ID, participants, data2 )
58         remote_counters[remote_ID] ← remote_counters[remote_ID] ∪ {(remote_seq_counter, current_tir

59
60   // simple multicast message.
61   else if msg_type = MSG_TYPE_SMCAST
62     var (participants, data) ← payload
63     multicast_deliver( source, participants, data )
64
65   // default handler for messages.
66   else
67     default_receive( source, msg_type, payload )
68 end
69
70 // parallel process. this is called periodically by a timer
71 // to clear old sequence numbers.
72 function cleanup( )
73 begin
74   for v ∈ V
75     sequence_numbers ← remote_counters[v]
76     for (seq, timestamp) ∈ sequence_numbers
77       if timestamp < current_time − 2δ
78         sequence_numbers ← sequence_numbers \{(seq, timestamp)}
79     remote_counters[v] ← sequence_numbers
80   end
81 end
```

Practical implementation concerns

If one wants to implement the algorithms shown in Listing 5.2 the following things should be considered.

- The variable ID is simply the MAC address of the first network interface of this

node. This provides a unique global address assuming that the network configuration is correct.

- Assuming a wireless network with an IEEE802.11 network layer the function ' lowlevel_multicast ' should be implemented by using the *msg_type* as the Ethernet frame type and by sending to the MAC broadcast address.
- The function ' lowlevel_receive ' should take all messages received by the MAC layer. If the message is either of the type 'MSG_TYPE_SMCAST' or 'MSG_TYPE_RMCAST' is should be processed by the appropriate functions. Otherwise, the default handler should be used.
- If node recovery is required the local sequence counter must be updated in a safe manner, for example, by using a local transaction spanning the entire multicast. Furthermore the remote sequence counters have to be restored on startup.

Implementation of the Thallner NBAC

Having proved that such a reliable multicast primitive can be implemented we can use the synchronous NBAC algorithm from Raynal to implement the Thallner algorithm. This is shown in Listing 5.3. The correctness of the NBAC is shown in [Ray96, p.11-p.13]. The finalization concept is easy in a synchronous system because it simple has to ensure that every participant has executed the decision.

Definition 35 (Finalization)**.** *A protocol solving NBAC finalization in the Thallner sense has the following properties:*

Local Finalization: *Eventually 'finalize_IMPL' will be called on each correct participant after 'decision_IMPL' has terminated on this participant.*

Finalization Agreement: *No two participants decide on different finalize results.*

Finalization Validity: *If a participant decides on the finalize result* ***COMMIT****, then decision has terminated on every participant.*

Finalization Non-Triviality: *If there is no failure suspicion, then the finalize result is* ***COMMIT****.*

Theorem 6. *The algorithm shown in Listing* 5.3 *solves the finalization problem.*

Proof. Finalization agreement is simple because we use the same result value as for the NBAC. Since the NBAC guarantees uniform agreement we get the finalization agreement for free. The *finalization validity* property is easy to achieve in a synchronous system because the execution of statements can differ by at most the time δ at any node if we can neglect computation times. The reason for this is that the only time which depends on the network is the reception of the initial multicast message. Therefore the property can easily be implemented by waiting the time δ in line 49. *Finalization non-triviality* is also simple. If all nodes have sent their votes and their votes are **COMMIT**, then the

result is **COMMIT** and so is the finalize result. *Local finalization* is simple because the code is sequential and the execution of 'finalization_IMPL' is after 'decision_IMPL'. □

Listing 5.3: NBAC protocol for the TBUT network model

```
1  function initiate( participants, data )
2  begin
3    // used to run multiple instances of the NBAC protocol at once.
4    var nbac_uid ← global_unique_id( )
5    // send multicast with the data to vote upon.
6    multicast_send( participants, (nbac_uid, data) )
7  end
8
9  function multicast_deliver( source, participants, data )
10   // extract the data from the multicast message payload
11   var (nbac_uid, nbac_data) ← data
12   nbac(nbac_uid, participants, nbac_data )
13 end
14
15 function s_rel_multicast_deliver( source, participants, data )
16  // extract the data from the multicast message payload
17   var (nbac_uid, vote, commit_data) ← data
18   vote(nbac_uid, source) ← (vote, commit_data)
19 end
20
21 function nbac( nbac_uid, participants, data )
22 begin
23   // let node vote on the data using the application dependent
24   // function vote_IMPL.
25   var (vote, commit_data) ← vote_IMPL(nbac_uid, data)
26   // synchronous reliable multicast requires time δ'
27   s_rel_multicast_send( participants, (nbac_uid, vote, commit_data) )
28   set timer = create_timer(δ + δ')
29   wait for
30     // all votes have been received
31     ∀v ∈ participants : vote(nbac_uid, v) = COMMIT
32     // one participant wants to abort
33     ∨ ∃v ∈ participants : vote(nbac_uid, v) = ABORT
34     // timer has expired
35     ∨ expired(timer)
36
37   if ∃v ∈ participants : vote(nbac_uid, v) = ABORT then
38     var result ← ABORT
39   if expired(timer) then
40     var result ← ABORT
41   if ∀v ∈ participants : vote(nbac_uid, v) = COMMIT then
42     var result ← COMMIT
43
44   var global_commit_data ←
       {commit_data : ∀v ∈ participants(∃(vote, commit_data) ∈ vote(nbac_uid, v))}
```

```
45    decision_IMPL(nbac_uid, result, data, global_commit_data)
46
47    // wait until every correct node has executed decision_IMPL
48    set timer = create_timer(δ)
49    wait for expired(timer)
50    finalize_IMPL(nbac_uid, result, data, global_commit_data)
51  end
```

Theorem 7. *The total message complexity of the Thallner NBAC for n participants is in $O(n^2)$.*

Proof. Initially the initiator initiates the NBAC by sending a simple multicast message to all participants. Using our multicast implementation this accounts for 1 message. Every node which participates in the multicast must multicast its vote to all other nodes using a reliable multicast. One reliable multicast requires n messages and therefore n reliable multicasts require n^2 message. In total we have $n^2 + 1$ messages which is $O(n^2)$. □

5.3.2 Implementation using Agreement

Our second algorithm only uses a simple multicast and uses an additional agreement phase. Using our network model it allows a very efficient implementation which has a message complexity of $O(n)$. We start by replacing the ' s_rel_multicast_send ' primitive with the normal multicast primitive. The resulting algorithm is shown in Listing 5.4. Warning - The algorithm below should *NOT* be used for anything because it violates the uniform agreement property and is only shown here for explanation purposes.

Listing 5.4: Bad NBAC protocol for the TBUT network model

```
1  function initiate( participants, data )
2  begin
3    // used to run multiple instances of the NBAC protocol at once.
4    var nbac_uid = global_unique_id( )
5     // send multicast with the data to vote upon.
6    multicast_send( participants, (MSG_TYPE_INITIATE, nbac_uid, data) )
7  end
8
9  function multicast_deliver( source, participants, data )
10   // extract the msg type from the multicast message payload
11   var (msg_type, ...) ← data
12   if msg_type = MSG_TYPE_INITIATE
13     // now we know the type and can extract all information.
14     var (msg_type, nbac_uid, nbac_data) ← data
15     nbac(nbac_uid, participants, nbac_data )
16   else if msg_type = MSG_TYPE_VOTE
17    // now we know the type and can extract all information.
18    var (msg_type, nbac_uid, vote, commit_data) ← data
19    vote(nbac_uid, source) ← (vote, commit_data)
20 end
21
```

```
22 function nbac( nbac_uid, participants, data )
23 begin
24   // let node vote on the data using the application dependent
25   // function vote_IMPL.
26   var (vote, commit_data) ← vote_IMPL(nbac_uid, data)
27   multicast_send( participants, (MSG_TYPE_VOTE, nbac_uid, vote, commit_data) )
28   set timer = create_timer(2δ)
29   wait for
30     // all votes have been received
31     ∀v ∈ participants : vote(nbac_uid, v) = COMMIT
32     // one participant wants to abort
33     ∨ ∃v ∈ participants : vote(nbac_uid, v) = ABORT
34     // timer has expired
35     ∨ expired(timer)
36
37   if ∃v ∈ participants : vote(nbac_uid, v) = ABORT then
38     var result ← ABORT
39   if expired(timer) then
40     var result ← ABORT
41   if ∀v ∈ participants : vote(nbac_uid, v) = COMMIT then
42     var result ← COMMIT
43
44   // synchronize with other nodes. this line is executed at
45   // different nodes at most δ apart.
46   wait for expired(timer)
47
48   var global_commit_data ←
         {commit_data : ∀v ∈ participants(∃(vote, commit_data) ∈ vote(nbac_uid, v))}
49   decision_IMPL(nbac_uid, result, data, global_commit_data)
50
51   // wait until every correct node has executed decision_IMPL
52   set timer = create_timer(δ)
53   wait for expired(timer)
54   finalize_IMPL(nbac_uid, result, data, global_commit_data)
55 end
```

An example which violates the uniform agreement is easily given. Assume that there are 4 nodes, where Figure 5.1 shows the transmission graph of the network.

1. Node 1 initiates the NBAC protocol and sends a message to the participants $1, 2, 3$ and 4.
2. Node 1 receives the message and votes **COMMIT** and broadcasts its vote.
3. Node $2, 3$, and 4 also receive the message and vote **COMMIT**.
4. Node 1 would receive all votes and therefore would vote **COMMIT**. The same holds for node 3.
5. Node 2 would miss the vote from node 4 and node 4 the vote from node 2. Therefore, these votes would be **ABORT**.

6. We now have nodes 1 and 3 voting **COMMIT** and nodes 2 and 4 voting **ABORT**. This violates the *uniform agreement* property.

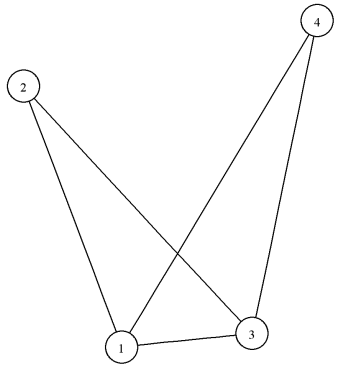


Figure 5.1: Counterexample for modified NBAC.

Therefore, we have chosen to add an additional and optimized agreement protocol which assures that the variable *result* is consistent among all nodes. The basic structure is shown in Listing 5.5. The algorithm chosen for our agreement, which is shown in Listing 5.6, can tolerate f failures, where $f \leq n-2$, and requires 2 synchronous rounds. In each round, every node broadcasts its current set of votes to all nodes. After 2 rounds, the nodes choose the minimum value from their current set, which is taken as the input to the decision. We will show the complete algorithm in the following section together with its correctness proof.

Listing 5.5: NBAC protocol with agreement for the TBUT network model

```
1  function initiate( participants, data )
2  begin
3    // used to run multiple instances of the NBAC protocol at once.
4    var nbac_uid = global_unique_id( )
5    // send multicast including the participants and the data to
6    // vote upon.
7    multicast_send( participants, (MSG_TYPE_INITIATE, nbac_uid, data) )
8  end
9
10 function multicast_deliver( source, participants, data )
11   // extract the msg type from the multicast message payload
12   var (msg_type, ...) ← data
13   if msg_type = MSG_TYPE_INITIATE
14     // now we know the type and can extract all information.
15     var (msg_type, nbac_uid, nbac_data) ← data
16     nbac(nbac_uid, participants, nbac_data )
17   else if msg_type = MSG_TYPE_VOTE
18     // now we know the type and can extract all information.
19    var (msg_type, nbac_uid, vote, commit_data) ← data
20    vote(nbac_uid, source) ← (vote, commit_data)
```

```
21    else
22      // pass to other handlers
23  end
24
25  function nbac( nbac_uid, participants, data )
26  begin
27    // init function for agreement algorithm. described later.
28    preinit(nbac_uid)
29    // let node vote on the data using the application dependent
30    // function vote_IMPL.
31    var (vote, commit_data) ← vote_IMPL(nbac_uid, data)
32    multicast_send( participants, (MSG_TYPE_VOTE, nbac_uid, vote, commit_data) )
33    // during two δ the clocks of the nodes can drift at most
34    // 2J apart.
35    set timer = create_timer(2δ + 2J)
36    wait for
37      // all votes have been received
38      ∀v ∈ participants : vote(nbac_uid, v) = COMMIT
39      // one participant wants to abort
40      ∨ ∃v ∈ participants : vote(nbac_uid, v) = ABORT
41      // timer has expired
42      ∨ expired(timer)
43
44    if ∃v ∈ participants : vote(nbac_uid, v) = ABORT then
45      var local_result ← ABORT
46    if expired(timer) then
47      var local_result ← ABORT
48    if ∀v ∈ participants : vote(nbac_uid, v) = COMMIT then
49      var local_result ← COMMIT
50
51    // synchronize with other nodes.
52    wait for expired(timer)
53
54    // again the UID is used to allow multiple instances.
55    var agreed_result = agreement(nbac_uid, participants, result)
56
57    // agreement finished. now execute the decision.
58    var global_commit_data ←
          {commit_data : ∀v ∈ participants(∃(vote, commit_data) ∈ vote(nbac_uid, v))}
59    decision_IMPL(nbac_uid, agreed_result, data, global_commit_data)
60
61    // wait until every correct node has executed decision_IMPL.
62    // clocks have already drifted up to 6J and we wait for another
63    // tick δ.
64    set timer = create_timer(δ + 7J)
65    wait for expired(timer)
66    finalize_IMPL(nbac_uid, agreed_result, data, global_commit_data)
67  end
```

Lemma 2. *If all nodes are correct and no link has failed, then all nodes have the same value for local_result after line* 49*. Let* $local\_result_{p_i}$ *denote the result for participant* $p_i \in P$*, then* $\forall p_i, p_j \in P : (local\_result_{p_i} = local\_result_{p_j})$.

Proof. If no node and no link has failed, every node will execute the function in line 25 at time $t \leq \delta$ because of our network assumptions and the fact that no message was lost. Every node will send its vote, and all votes will be received by every node in line 10 and will be added to the set of votes in line 20. At most at time $t \leq 3\delta$ all nodes have passed line 49 and have chosen their local result. Because all nodes have the same messages, the result is the same. □

Theorem 8. *If at most* $f \leq n - 2$ *nodes have failed and a node has crashed before sending its vote in line* 32 *then this fact is detected by at least one correct node.*

Proof. Assume a node has crashed before sending its vote. Since $f \leq n - 2$ there are always two correct nodes and these nodes will detect the missing vote and will set their local result to **ABORT** in line 46. □

Please note that up to now we have not enforced a consistent view for the network, which will be taken care of in the second part of our algorithm.

Agreement protocol

We will now show how this can be achieved by using our agreement protocol. The variable V holds the exchanged votes. The variable *queue* is a global variable and holds the round k messages for a given NBAC instance. The variable *round* holds the current active round for a given NBAC.

Listing 5.6: Agreement protocol in the TBUT network model

```
1  var queue
2  var round
3
4  procedure preinit(agreement_uid)
5  begin
6    round(agreement_uid) = 1
7  end
8
9  procedure multicast_deliver( source, participants, data )
10 begin
11   // extract the msg type from the multicast message payload
12   var (msg_type,...) ← data
13   if msg_type = MSG_TYPE_AGREEMENT
14     // extract the complete message after type is known
15     (msg_type, agreement_uid, V) ← data
16     queue(source, agreement_uid, round(agreement_uid)) ← V
17 end
18
19 procedure agreement(agreement_uid, participants, local_result)
```

```
20 begin
21   var V ← {local_result}
22   for k, 1 ≤ k ≤ 2
23     multicast_send( participants, (MSG_TYPE_AGREEMENT, agreement_uid, V) )
24     // clocks have already drifted up to 2J. In every round
25     // the can drift another 2J relatively.
26     set timer = create_timer(2δ + 2J + 2Jk))
27     wait for expired(timer)
28     for v ∈ participants
29       if queue(v, agreement_uid, round(agreement_uid)) ≠ ⊥
30         V = V ∪ queue(v, agreement_uid, round(agreement_uid))
31     round(agreement_uid) = k
32
33   var result ← COMMIT
34   if ABORT ∈ V
35     result ← ABORT
36   return result
37 end
```

First we note that all nodes will start at most δ apart if 'agreement' algorithm is executed from within Listing 5.5. The reason is that the only time which depends on the network is the initial reception of the multicast message from the originator. Therefore, the execution is aligned on grids, which is shown in Figure 5.2. Note that there are always overlapping areas for every round. The carefully reader will notice that enforcing this grid actually requires our network model assumption shown in Definition 34, that the local clocks are not allowed to drift unbounded to each other within a given time frame δ. This fact has been taken into account by the maximum clock jitter J. The time required to wait increases with every new round because of this jitter.

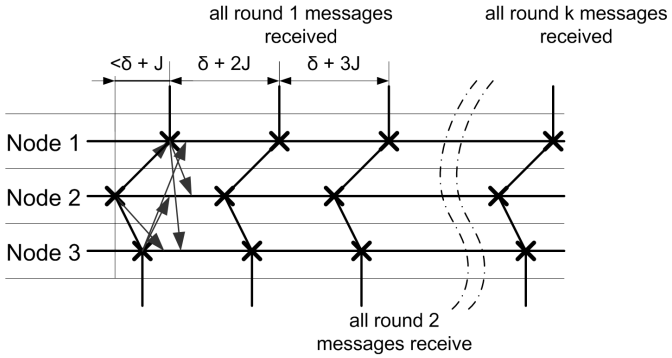


Figure 5.2: Execution schedule for agreement algorithm

An agreement protocol is defined by having the properties shown in Definition 36 based on [AW04, p.95].

Definition 36 (Agreement protocol)**.** *Let $V = \{v_1, \ldots, v_n\}$ be a set of votes and let v_i be the vote of node p_i. The values v_i are either **ABORT** or **COMMIT** in our case. At*

the end of the algorithm every node chooses a value y_i from the proposed ones.

Termination: *Every correct node p_i eventually assigns a value to y_i.*

Uniform Agreement: *No two processes (correct or not) decide on different values.*

Validity: *If $v_i = v$ for all votes $v_i \in V$ then $y_i = v$ for any non-faulty process p_i after execution.*

Theorem 9. *The algorithm in Listing 5.6 solves consensus in our modified TBUT network model and tolerates up to $f \leq n - 2$ link or node failures.*

Proof. **Termination:** The termination property is simple because the system is synchronous and it does not wait for external events. Therefore, every non-crashed process which started the algorithm at time t returns from the function at the time $t + 2 \cdot \delta$.

Validity: Validity is trivial because a process adds its own local result to the set V at the beginning in line 21. Because it sends only values from this set and receives only values from processes participating in this instance, the only possible values to choose from are the initially proposed ones.

Uniform Agreement: Again we can invoke Lemma 1 because of the similarity of the algorithms. The broadcasting of the message is implemented by queuing the received message locally in line 16 and forwarding it again at the beginning of the next round. The delivery is adding the received votes to the set. At the beginning every node has its own vote in its local set V. This set is forwarded and received by other nodes in round 1. At least some correct nodes will receive this set and will forward it again. Therefore by Lemma 1 all processes end up with the same set V and will decide on the same value. Crashed processes do not decide at all and therefore *Uniform Agreement* holds.

□

Combining the agreement protocol and modified NBAC protocol allows us to provide an implementation for the NBAC service which is required by the TMA.

Theorem 10. *Combining the algorithm shown in Listing 5.5 together with the uniform agreement algorithm in Listing 5.6 solves the NBAC problem.*

We have already shown in Theorem 2 that our basic Algorithm in Listing 5.5 fulfills the *non-triviality* property. *Termination* is simple because the algorithm does not wait infinitely long. *Integrity* is obvious by the algorithm itself. *Validity* follows from the fact that a node will only set its *local_result* to **COMMIT** if all nodes have sent their votes and these votes are **COMMIT**. Because the uniform agreement algorithm only returns a result which has been an input value, this desired property follows easily. *Uniform agreement* is a direct result of the uniform agreement algorithm.

Theorem 11. *The combined algorithm sends* $1 + n + 2 \cdot n$ *messages in total and has therefore a complexity of* $O(n)$*.*

Proof. Initially the initiator sends a single message. Then every node must send its vote which requires n messages in total. The modified agreement algorithm requires $2 \cdot n$ messages. This accounts for a total of $1 + n + 2 \cdot n$ messages. □

A Installation

Installation of NS2 is a difficult process because it requires a lot of different modules. The easiest way to get started is to use the NS2 all-in-one package in version 2.29, available on the NS2 website `http://www.isi.edu/nsnam/ns/`. We will only cover the installation on Windows XP using a Cygwin environment, but most of this information directly applies to any UNIX variant unless otherwise noted.

A.1 Required components

This list shows all required components regardless if a development or simulation environment is built.

- A recent Cygwin ($> 1.5.xx$) with a recent GCC compiler ($> 3.4.4$). The complete Cygwin distribution is available on `http://www.cygwin.com`.
- The C++ boost library (> 1.33). It is available on `http://www.boost.org`.
- The NS2 all-in-one release `ns-allinone-2.29.tar.gz` with the MD5 hash `4cf7-f984253634c16ad7dc50b2342db9`.
- The NS2 thallner patch `ns2-thallner-1.8.patch` with the MD5 hash `1370daa2-0beac7c1e250de1fee4284de` available from the author of this thesis.
- The NS2 SSF patch `ssf-1.7.patch` with the MD5 hash `8505b5406fd8dd514ef6-bcc1ea79cc25` for the *source-sequenced flooding* protocol available from the author of this thesis. This is optional and is only required if this protocol should be used.
- The NS2 SSF-thallner combo patch `ns2-thallner-1.8-ssf-1.7-combo.patch` with the MD5 hash `75866988093e38ca8584d3cd5cd5fab1`. This patch combines the two previous patches and is easier to apply.

The next list contains additional components which should be installed if debugging support is needed. Debugging is almost always needed when modifications to the TMA are made.

- The TCL debugging component `tcl-debug-2.0-pmsrve.tar.gz` with the MD5 hash `38d4ff649a4b990cbcd49e7d47ae5d57` from Pedro Vale Estrela. The modified version supports the 'c' command to continue up to the following breakpoint and the 'C' command for run to completion. Furthermore, it adds history support using the up/down keys.

- The patch for the TCL debugging component to fix a problem with memory allocation. The patch is available as `tcl-debug-2.0-pmsrve-ckalloc.patch` with the MD5 hash `ddd01c7bf3922a9c17d42952bd41fae1`. It is available on request from the author of this thesis.

- If debugging of memory leaks is required, the dmalloc library from `http://dmalloc.com`. The author used version 4.8.2.

A.2 Installation of NS2

We start by extracting the NS2 all-in-one package and by applying the required patches to the core distribution. First you should start a Cygwin shell and switch to an appropriate directory where you want to install all NS2 components. In our example, we assume that this directory is `/opt/thallner`. If you use a different one, make sure that you change all references to this directory accordingly. Furthermore we assume that all required components are available in the subdirectory `files` of `/opt/thallner`.

If you only want to apply the Thallner patches, use the patch `ns2-thallner-1.8.patch`. If you only want to use the SSF patch, use the patch `ns2-ssf-1.7.patch`.

```
$ gzip -dc files/ns-allinone-2.29.tar.gz | tar -xf -
...
$ # move unpacked files up one directory hierarchy
$ mv ns-allinone-2.29/* . && rmdir ns-allinone-2.29
$ cd ns-2.29
$ patch -p1 < ../files/ns2-thallner-1.8-ssf-1.7-combo.patch
patching file Makefile.in
patching file common/packet.h
...
patching file trace/cmu-trace.cc
patching file trace/cmu-trace.h
...
$ cd ..
```

Now the build should be started by executing the install script. Depending upon your system's computing power, this will take about 1hour.

```
$ ./install
==========================================================
* Testing for Cygwin environment
==========================================================
Cygwin detected
...
$
```

After this step, you should have a working installation of NS2 with the required patches applied.

A.3 Installation of supporting utilities

The following tools should be installed to make full use of our framework.

- GraphViz is a graph drawing software available from `http://www.graphviz.org/`. We recommend using a version greater than 2.12. This software is required to process the topology graphs and topology trees.
- If you need to generate EPS output, the author recommends the tool `sam2p` available from `www.inf.bme.hu/~pts/sam2p/`. It can be used to convert PNG output files from GraphViz into EPS images by calling

  ```
  sam2p input.png -2 output.eps
  ```

A.4 Testing of installation

The functionality of NS2 can be tested by the validation suite. This is the first step and validates if there are any problems with NS2. Please note that NS2 2.29 has some known problems under Cygwin and not all tests will pass. It can be executed by the following commands

```
$ cd /opt/thallner/ns-2.29
$ ./validate
...
$
```

The next thing to test is the SSF protocol (if used). For this purpose, a small test NS2 script should be executed.

```
$ cd /opt/thallner/ns-2.29
$ cd tcl/test
$ ../../ns test-suite-ssf.tcl
num_nodes is set 8
creating node 0 in group = 0
...
Sequence number statistics for node 7:
source | seq_num | sent | received | droped | forwarded | last
     0 |       6 |    0 |        0 |     12 |         6 | 11.00748
     7 |       2 |    1 |        0 |      3 |         1 | 13.00874
Sequence number statistics for node 2:
source | seq_num | sent | received | droped | forwarded | last
     0 |       6 |    0 |        0 |     12 |         6 | 11.00748
     2 |       1 |    0 |        0 |      0 |         0 | 0.00000
     7 |       1 |    0 |        0 |      0 |         1 | 13.00334
Sequence number statistics for node 0:
source | seq_num | sent | received | droped | forwarded | last
     0 |       7 |    6 |        0 |     11 |         6 | 11.00529
     7 |       1 |    0 |        0 |      0 |         1 | 13.00487
$
```

Finally, the Thallner protocol should be tested by executing the commands shown below. After execution, the simulated topology is available in `test-suite-thallner-topology-graph-t92-neato.dot` and `test-suite-thallner-topology-tree-t92-dot.dot`. These files can then be processed by GraphViz.

```
$ cd /opt/thallner/ns-2.29
$ cd tcl/test
$ ../../ns test-suite-thallner.tcl
num_nodes is set 20
INITIALIZE THE LIST xListHead
channel.cc:sendUp - Calc highestAntennaZ_ and distCST_
highestAntennaZ_ = 1.5, distCST_ = 406.3
SORTING LISTS ...DONE!
...
node 0 has changed its group structure at 12.086313
node 1 has changed its group structure at 15.555068
node 2 has changed its group structure at 0.667502
node 3 has changed its group structure at 0.000000
...
$
```

B Development support

Adding new software components to NS2 requires additional setup steps to the ones described in Appendix A.

B.1 Required components

- A C/C++ compiler. The setup was tested with versions > 3.4.4.
- The Insight GDB Debugger frontend available from `http://sourceware.org/insight/`.
- Diff and patch utilities for the Cygwin environment to create patches.
- Doxygen available from `http://www.doxygen.org` to create source code documentation.
- AStyle - Artistic Style to reformat the source code. It is available from `http://astyle.sourceforge.net/`.

B.2 File system layout

This section refers to the complete development tree available as a separate tar-ball `ns2-sourcetree-xxx.tar.gz` from the author of this thesis. It contains additional scripts to support development.

files/ Contains all required source files. These files have not been modified and can always be replaced by original ones obtained from the appropriate vendor.

ns-2.29.orig/ A copy of the directory `ns-2.29` from the `ns-allinone-2.29.tar.gz` tar-ball. This is used to generate patches for NS2.

ns2-ssf/ SSF - *Source-Sequenced-Flooding* protocol implementation.

ssf/ Contains most of the implementation of the SSF protocol. The packet format is described in `hdr_ssf.h` and the main implementation of the routing protocol can be found in `ssf.cc` and `ssf.h`.

update-thallner-rep.sh This shell script can be used to update an original source directory with the required files for the SSF protocol. It should be executed as

```
\shell{ns2-ssf/update-thallner-rep.sh ns2-ssf ns-2.29}
```

which copies all required files from the `ns2-ssf` directory into the `ns-2.29` directory. The updated repository can then be used to generate a patch as described in B.4.

ssf-x.x.patch Patches to NS2 created by the `genpatch.sh` utility.

others The other files are C++ and TCL source files are modifications of existing NS2 files and are based on version 2.29.

ns2-thallner/ Implementation of the Thallner algorithm.

mac/ The core interface of the Thallner algorithm. The files `tma.h` and `tma.cc` provide an abstract base class for all TMA implementations. An example and very simple implementation of a topology management algorithm is shown in `tma-filter.cc` and `tma-filter.h`.

thallner/ Implementation of the Thallner algorithm together with all required components.

update-thallner-rep.sh This shell script can be used to update an original source directory with the required files for the Thallner algorithm. It should be executed as

```
ns2-thallner/update-thallner-rep.sh ns2-thallner ns-2.29
```

which copies all required files form the `ns2-thallner` directory into the `ns-2.29` directory. The updated repository can then be used to generate a patch as described in B.4

ns2-thallner-x.x-ssf-x.x-combo.patch Combo patches for SSF and Thallner because of patch conflicts.

ns2-thallner-x.x.patch Patches to NS2 created by the `genpatch.sh` utility.

statistics/ Contains a program to combine multiple Excel output files into a single output file. See the scripts `merge.sh` and `merge-all.sh` on how to use this program.

scripts/ Simple utilities

build-ns2.sh This script should be executed from within the NS2 source directory. It calls configure with the appropriate arguments and builds a working NS2 version.

fixup-tree.sh This script strips TCL and C++ files of Windows CR/LF line feeds. This script should always be executed before committing CVS changes.

genpatch.sh This script generates patches. It is called as `genpatch.sh ORIG MODIFIED` and outputs the diff on stdout.

reformat.sh Calls the `astyle` utility to reformat the source code. Should be called on any source file NOT from the ns2 distribution[1]

[1] If called on NS2 source files the diffs are large because NS2 does not follow any coding conventions.

B.3 Adding new functions

In this example, we assume that a new function should be added to the Thallner algorithm. We first start by copying an original NS2 source tree and by applying the Thallner patches to this directory.

```
$ cd /opt/thallner
$ rm -rf ns-2.29
$ cp -Rpf ns-2.29.orig ns-2.29
$ ./ns2-thallner/update-thallner-rep.sh ns2-thallner ns-2.29
copying ns2-thallner/common/packet.h to ns-2.29/common/packet.h...
copying ns2-thallner/mac/tma-filter.cc to ns-2.29/mac/tma-filter.cc
    ...
copying ns2-thallner/mac/tma-filter.h to ns-2.29/mac/tma-filter.h...
copying ns2-thallner/mac/tma.cc to ns-2.29/mac/tma.cc...
...
$
```

We can now start a build of the source tree by executing the following commands:

```
$ cd ns-2.29
$ ../scripts/build-ns2.sh
No .configure file found in current directory
Continuing with default options...
checking build system type... i686-pc-cygwin
...
$ make
/cygdrive/c/Projects/adhoc/ns2/bin/tclsh8.4 bin/tcl-expand.tcl tcl/
    lib/ns-lib.tc
l tcl/lib/ns-diffusion.tcl  | ../tclcl-1.17/tcl2c++ et_ns_lib > gen/
    ns_tcl.cc
...
$ file ns
ns: MS-DOS executable PE  for MS Windows (console) Intel 80386 32-bit
$
```

You can now make changes to any files. If a new source file is added make sure that you have added it to the makefile `Makefile.in` in the root directory of NS2 to include it in the build. After this you should run a build using the following commands.

```
$ make depend
...
$ make
...
$
```

B.4 Creating a patch

Creating a patch suitable for distribution is not an easy task. The following things should be considered every time when creating a new patch.

- Always create the patch against an original version of NS2. By convention, we assume that in our setup an original source tree is available in `ns-2.29`.
- We recommend having a separate directory tree, which contains all files, for every module. For example see `ns2-ssf` and `ns2-thallner`.
- We also recommend having a script like `update-thallner-rep.sh` in every directory, which can be used to copy the files.

Let us assume that we have a modified version of the Thallner algorithm in the directory `ns-2.29`, the original source tree in `ns-2.29.orig`, and a directory for the Thallner source files in `ns2-thallner`. The first directory, `ns-2.29`, is our temporary working directory used during development. The `ns2-thallner` directory contains all files for this module (it includes new files and modified NS2 source files). First, we start by merging the updated source files back with our master directory.

```
$ cd /opt/thallner
$ ./ns2-thallner/update-thallner-rep.sh ns-2.29 ns2-thallner
copying ns-2.29/common/packet.h to ns2-thallner/common/packet.h...
copying ns-2.29/mac/tma-filter.cc to ns2-thallner/mac/tma-filter.cc
    ...
copying ns-2.29/mac/tma-filter.h to ns2-thallner/mac/tma-filter.h...
copying ns-2.29/mac/tma.cc to ns2-thallner/mac/tma.cc...
...
$
```

Now we can remove the `ns-2.29` directory and create a clean copy of it by merging in the modifications again. This step is required because the working directory contains a lot more files created during the build which should not be in the final patch.

```
$ cd /opt/thallner
$ rm -rf ns-2.29
...
$ cp -Rpf ns-2.29.orig ns-2.29
$ ./ns2-thallner/update-thallner-rep.sh ns2-thallner ns-2.29
copying ns-2.29/common/packet.h to ns2-thallner/common/packet.h...
copying ns-2.29/mac/tma-filter.cc to ns2-thallner/mac/tma-filter.cc
    ...
copying ns-2.29/mac/tma-filter.h to ns2-thallner/mac/tma-filter.h...
...
```

Next we generate the patch by calling

```
$ ./scripts/genpatch.sh ns-2.29.orig ns-2.29 > ns2-thallner/ns2-
    thallner-1.x.pa
tch
```

The next step is optional, but it is recommended by the author of this thesis to test the patch.

```
$ cd /opt/thallner
$ cp -Rpf ns-2.29.orig ns-2.29.testing
```

```
$ cd ns-2.29.testing
$ patch -p1 < ../ns2-thallner/ns2-thallner-1.x.patch
patching file Makefile.in
patching file common/packet.h
patching file conf/configure.in.head
patching file configure
patching file html/dox.css
patching file html/dox_html_footer
...
$ ../scripts/build-ns2.sh
No .configure file found in current directory
Continuing with default options...
checking build system type... i686-pc-cygwin
checking host system type... i686-pc-cygwin
...
$ make
...
```

B.4.1 Documentation

Source code documentation is generated by Doxygen. Doxygen is a source code documentation generator tool and is freely available. The main configuration can be found in `html/doxygen.conf` within the `ns2-thallner` directory. Assuming an appropriatly patched NS2 source directory, the documentation can be generated by:

```
$ cd /opt/thallner/ns-2.29
$ cd html
$ doxygen doxygen.conf
Searching for include files...
Searching for example files...
Searching for images...
...
```

The output is generated in the subdirectory `html`.

C Usage

We will not cover all details of using the NS2 simulator. Instead, we will focus on the parts which are important for our simulation. For more information, the interested reader is referred to the excellent documentation in [FV06].

C.1 Introduction

We have provided a template suitable for our studies. It is available from the contact author and is located in the directory `scripts/template`. It consists of three components. A TCL script for generating network topologies described in Section C.2, a NS2 simulator setup script shown in Section C.3, and a Makefile. We start by describing the Makefile, which is shown in Listing C.1. The Makefile is used to start the simulation, generate a new network topology, and to convert the GraphViz neato and dot files into PNG and EPS images. Other (non-essential) options are the ability to invoke the Insight debugger from the command line or to remove any temporary files generated during the simulation.

Listing C.1: Makefile for simulation framework

```
# ------------------------------------------------------------------------
# Author: Walter Christian
#
# $Id: usage-makefile,v 1.2 2007/08/22 19:37:46 cwalter Exp $
#
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# path to binary executable
# ------------------------------------------------------------------------
NS2_DIR   = ../../ns2/ns-2.29
NS2       = $(NS2_DIR)/ns
INSIGHT   = /opt/insight-x86/bin/insight
NEATO     = neato
DOT       = dot
SAM2P     = sam2p

# ------------------------------------------------------------------------
# project dependent scripts
# ------------------------------------------------------------------------
NS2_SCRIPT      = adhoc.tcl
NS2_GEN_SCRIPT = adhoc-gen.tcl
NS2_TRACE       = adhoc.tr
```

```
NUM_NODES       = 10
GW_NODES        = 3
THALLNER_K      = 3
VARIANT         = a
PREFIX          = n$(NUM_NODES)-gw$(GW_NODES)-$(VARIANT)
# ------------------------------------------------------------------------
# targets
# ------------------------------------------------------------------------

all: trace

network:
  $(NS2) $(NS2_GEN_SCRIPT) $(NUM_NODES) $(GW_NODES) $(VARIANT)

trace:
  $(NS2) $(NS2_SCRIPT) $(NUM_NODES) $(GW_NODES) $(THALLNER_K) $(VARIANT)

trace-dbg:
  $(INSIGHT) --args $(NS2) $(NS2_SCRIPT) $(NUM_NODES) $(GW_NODES) $(
      THALLNER_K) $(VARIANT) 2>&1


images:
  @for f in `find . -maxdepth 1 -name "*neato*.dot"`; do \
      $(NEATO) -Tpng -o $${f%dot}png $$f; \
  done
  @for f in `find . -maxdepth 1 -name "*dot*.dot"`; do \
      $(DOT) -Tpng -o $${f%dot}png $$f; \
  done
  @for f in `find . -maxdepth 1 -name "*png"`; do \
      $(SAM2P) $$f -2 $${f%png}eps; \
  done

clean:
  $(RM) -f *neato*.dot *neato*.png *neato*.eps
  $(RM) -f *dot*.dot *dot*.png *dot*.eps
  $(RM) -f $(NS2_TRACE)
```

The Makefile supports the following targets described below.

network: Generates a new network transmission graph. This is described in depth in Section C.2. The resulting topology is stored in `n$(NUM_NODES)-gw$(GW_NODES)--$(VARIANT)-topo.tcl`.

trace: Executes the NS2 simulator using the generated topology.

trace-dbg: The same as the trace target but executes the simulator within the debugger.

images: Converts the GraphViz output files into PNG files. All files having the letters *neato* in their name are processed with the neato tool from the GraphViz suite. The files having *dot* in their name are processed with the dot tool.

clean: Removes output files created during simulation and by the images target.

C.2 Network Topology Creation

The topology generation script `adhoc-gen.tcl` generates up to num_nodes normal nodes and up to num_gw_nodes gateway nodes. It can either be started from the Makefile by executing the `network` target or by executing the following command line

```
$ ns adhoc-gen.tcl ${NUM_NODES} ${GW_NODES} ${VARIANT}
```

The generated script contains TCL commands used to create and position the nodes in the $100m \times 100m$ grid. This script is then sourced by the main one. The script is shown in Listing C.2.

Listing C.2: TCL script for topology creation

```
1  #---------------------------------------------------------------------
2  # Project: Toplogy Generation Template for NS2 simulation framework
3  # Author:  Christian Walter <e0225458@student.tuwien.ac.at>
4  #
5  #---------------------------------------------------------------------
6
7  #---------------------------------------------------------------------
8  # check for input arguments
9  #---------------------------------------------------------------------
10 if { $argc != 3 } {
11   puts "error: adhoc-gen.tcl requires three arguments!"
12   puts "usage: ns2 adhoc.tcl N_NODES N_GWNODES N_VARIANT"
13   exit 1
14 } else {
15   set topology_file "topologies/topo-n[lindex $argv 0]-gw[lindex $argv 1]-[
         lindex $argv 2].tcl"
16 }
17
18 #---------------------------------------------------------------------
19 # set configuration properties
20 #---------------------------------------------------------------------
21 set num_nodes            [lindex $argv 0]            ;# total nodes
22 set num_gw_nodes         [lindex $argv 1]            ;# total nodes
23
24 #---------------------------------------------------------------------
25 # Create network topology
26 #---------------------------------------------------------------------
27
28 set topology_fd [ open $topology_file w ]
29 set seed [ clock clicks ]
30 expr srand($seed)
31
32 for { set i 0 } { $i < $num_nodes } { incr i } {
33   puts $topology_fd "set node_($i) \[ \$ns node \]"
34   puts $topology_fd "\$node_($i) set X_ [ expr rand() * 100 ]"
35   puts $topology_fd "\$node_($i) set Y_ [ expr rand() * 100 ]"
36   puts $topology_fd ""
37 }
38
```

```
39 for { set i $num_nodes } { $i < ( $num_nodes + $num_gw_nodes ) } { incr i }
       {
40   puts $topology_fd "set node_($i) \[ \$ns node_ \]"
41   puts $topology_fd "\$node_($i) set X_ [ expr rand() * 100 ]"
42   puts $topology_fd "\$node_($i) set Y_ [ expr rand() * 100 ]"
43   puts $topology_fd "set tma_($i) \[ \$node_($i) set tma_(0) \]"
44   puts $topology_fd "\$tma_($i) gateway-node 1"
45   puts $topology_fd ""
46 }
47
48 close $topology_fd
```

An example output is shown in Listing C.3 where two normal nodes 0 and 1 (top) and one gateway node 12 (end) are shown. It is important to use the same topology when comparing results, and therefore this script should be executed only once and a backup of the topology should be made.

Listing C.3: Example topology created by `adhoc-gen.tcl`

```
1  set node_(0) [ $ns node ]
2  $node_(0) set X_ 90.615616035934352
3  $node_(0) set Y_ 76.658715948769228
4
5  set node_(1) [ $ns node ]
6  $node_(1) set X_ 3.0389509643609407
7  $node_(1) set Y_ 75.648858014330671
8
9  ...
10
11 set node_(12) [ $ns node ]
12 $node_(12) set X_ 81.204052260706234
13 $node_(12) set Y_ 96.506345689532509
14 set tma_(12) [ $node_(12) set tma_(0) ]
15 $tma_(12) gateway-node 1
```

C.3 Simulation Framework and Setup

Performing a simulation in NS2 requires the setup of the simulator. This includes at least setting the configuration properties in the script, the setup of the simulator objects, the creation of the traffic patterns, and a schedule defining the execution. If the end time of the simulation is not known a-priori, a convergence criterion has to be defined to end the simulation. All of these components with the exception of the traffic patterns are already available in the template `adhoc.tcl` shown in Listing C.4.

In lines $81-97$, the wireless node is configured. This is basically the same as the setup shown in the NS2 manual in [FV06, p144]. An important exception is the additional parameter tmaType for the node configuration which enables the TMA algorithm in line 85 from val(tma).
The configuration items in lines $30-46$ must match the settings in the topology creation script and the required simulation settings. The number of nodes should equal the

number of normal nodes plus the number of gateway nodes. The external topology file is sourced in line 198.
In lines 69 – 79, a simulator instance and the supporting objects are created. In most cases, these items should be left as they are, because they do not need to be configured. To end the simulation, three functions are provided, which are shown in lines 102 – 181. The function 'finish' flushes the trace file output, closes all output files, and quits the simulator. The function 'convergence−test' tests if the network has already converged. The function 'convergence−calculate−times' is a private function and should not be changed or modified. If no node has changed its group information, the current topology information is dumped and the simulation is ended. Otherwise, the test reschedules itself. Topology information can be dumped by the functions 'dump−topology−graph', which uses the node local information from all nodes to create a view of the topology, and the function 'dump−topology−tree', which outputs the topology tree. These functions are shown in lines 183 – 193.
The traffic patterns required for the simulation should be added after line 218. Additionally the simulation schedule shown in lines 223 – 224 should be modified to match the requirements of simulation. Finally, the traffic generators should be started. The TMA algorithm is started automatically at the nodes.

Listing C.4: Simulation Framework

```
1  #----------------------------------------------------------------------
2  # Project: Template for NS2 simulation framework
3  # Author:  Christian Walter <e0225458@student.tuwien.ac.at>
4  #
5  #----------------------------------------------------------------------
6
7  #----------------------------------------------------------------------
8  # check for input arguments
9  #----------------------------------------------------------------------
10 if { $argc != 4 } {
11   puts "error: adhoc.tcl requires four arguments!"
12   puts "usage: ns2 adhoc.tcl N_NODES N_GWNODES THALLNER_K N_VARIANT"
13   exit 1
14 } else {
15   set topo_num_nodes [lindex $argv 0]
16   set topo_num_gw_nodes [lindex $argv 1]
17   set topo_thallner_k [lindex $argv 2]
18   set topo_variant [lindex $argv 3]
19   set topo_prefix "n${topo_num_nodes}-gw${topo_num_gw_nodes}-${topo_variant
       }"
20   set topology_file "topo-${topo_prefix}.tcl"
21   if { ![file exists $topology_file] } {
22     puts stderr "error: the network topology $topology_file was not created
         "
23     exit 1
24   }
25 }
26
27 #----------------------------------------------------------------------
28 # set configuration properties
```

```
29 # ======================================================================
30   set num_nodes            100                          ;# total nodes
31 set xsize                  100
32 set ysize                  100
33
34 set val(chan)              Channel/WirelessChannel      ;# Channel Type
35 set val(prop)              Propagation/TwoRayGround     ;# radio-propagation
       model
36 set val(netif)             Phy/WirelessPhy              ;# network interface
       type
37 set val(mac)               Mac/802_11
38 set val(tma)               TMA/Thallner
39 set val(ifq)               Queue/DropTail/PriQueue      ;# interface queue type
40 set val(ll)                LL                           ;# link layer type
41 set val(ant)               Antenna/OmniAntenna          ;# antenna model
42 set val(ifqlen)            100                          ;# max packet in ifq
43 set val(rp)                DumbAgent
44
45 set last_stats_time       0.0
46
47
48 # ======================================================================
49 # Configure for IEEE802.11b
50 # ======================================================================
51 Mac/802_11 set SlotTime_           0.000020          ;# 20us
52 Mac/802_11 set SIFS_               0.000010          ;# 10us
53 Mac/802_11 set PreambleLength_     144               ;# 144 bit
54 Mac/802_11 set PLCPHeaderLength_  48                ;# 48 bits
55 Mac/802_11 set PLCPDataRate_      1.0e6             ;# 1Mbps
56 Mac/802_11 set dataRate_          11.0e6            ;# 11Mbps
57 Mac/802_11 set basicRate_         1.0e6             ;# 1Mbps
58
59 Phy/WirelessPhy set freq_        2.4e+9            ;# Frequency
60 Phy/WirelessPhy set Pt_          3.3962527e-2      ;# Transmission power
61 Phy/WirelessPhy set RXThresh_    6.309573e-12      ;# Receiver threshold
62 Phy/WirelessPhy set CSThresh_    6.309573e-12      ;# Sense threshold
63
64 TMA/Thallner set debug_ true
65
66 # ======================================================================
67 # Create a simulator instance
68 # ======================================================================
69
70 set ns [ new Simulator ]
71 set tracefd [ open $topo_prefix-k${topo_thallner_k}-adhoc.tr w ]
72 $ns use-newtrace
73 $ns trace-all $tracefd
74
75 set topo [ new Topography ]
76 $topo load_flatgrid $xsize $ysize
77
78 set god_ [ create-god $num_nodes ]
79 set chan_1_ [new $val(chan)]
80
```

```
81    $ns node-config \
82       -adhocRouting $val(rp) \
83       -llType $val(ll) \
84       -macType $val(mac) \
85       -tmaType $val(tma) \
86       -phyType $val(netif) \
87       -ifqType $val(ifq) \
88       -ifqLen $val(ifqlen) \
89       -antType $val(ant) \
90       -propType $val(prop) \
91       -topoInstance $topo \
92       -agentTrace ON \
93       -routerTrace OFF \
94       -macTrace OFF  \
95       -tmaTrace OFF \
96       -movementTrace OFF \
97       -channel $chan_1_
98
99  #-----------------------------------------------------------------
100 # Common functions
101 #-----------------------------------------------------------------
102   proc finish {} {
103     global ns tracefd
104     $ns flush-trace
105     close $tracefd
106     exit 0
107 }
108
109 proc convergence-calculate-times { arrname } {
110   global tma topo_thallner_k
111   upvar $arrname times
112
113   set num_nodes [ array size tma ]
114
115   # floor( (n-1)/(k-1) ) = ngroups is the number of groups. If the
116   # tree is balanced the height is log( ngroups ) / log( k ). Every
117   # node therefore walks up this hierarchy and creates group proposals
118   # where 1 is added because of the proposal for the node itself.
119   set group_tree_depth [ expr log( ( $num_nodes - 1 )/( $topo_thallner_k -
        1 ) ) / log( $topo_thallner_k ) + 1 ]
120
121   # add an extra factor of 2 because a generated proposal must also
122   # be accepted by the thallner algorithm.
123   set times(convergence_time_proposals) [ expr [ $tma(0) proposals-period ]
         * $group_tree_depth * 2.0 ]
124
125   # calculate the time required to check all groups at a node
126   set times(convergence_time_group) [ expr [ $tma(0) groupcheck-period ] *
        $group_tree_depth ]
127
128   # the convergence time is the maximum
129   if { $times(convergence_time_group) < $times(convergence_time_proposals)
        } {
130     set times(convergence_time) $times(convergence_time_proposals)
```

```
131   } else {
132     set times(convergence_time) $times(convergence_time_group)
133   }
134 }
135
136 proc convergence-test {} {
137   global ns tma val last_stats_time topo_prefix topo_thallner_k
138   set scheduler [ $ns set scheduler_ ]
139   set now [ $scheduler now ]
140
141   convergence-calculate-times times
142   stats-convergence-print stdout
143
144   # we assume that the network has converged.
145   set is_stable 1
146
147   # compute the time when the group structures have changed.
148   set last_change_max 0.0
149   for { set i 0 } { $i < [ array size tma ] } { incr i } {
150     set last_change [ $tma($i) last-change ]
151     if { $last_change > $last_change_max } {
152       set last_change_max $last_change
153     }
154     # if this node is not a gateway node it must have a node
155     # degree of k.
156     if { 0 == [ $tma($i) gateway-node ] } {
157       if { $topo_thallner_k != [ $tma($i) node-degree ] } {
158         set is_stable 0
159       }
160     }
161   }
162
163   # if the group structures have not changed during the convergence
164   # time the network is stable.
165   if { $now > $times(convergence_time) } {
166     if { $now - $last_change_max < $times(convergence_time) } {
167       puts "group_structure_has_changed_at_time_$last_change"
168       set is_stable 0
169     }
170   } else {
171     set is_stable 0
172   }
173
174   if { $is_stable == 1 } {
175     dump-topology-graph
176     dump-topology-tree
177     $ns at [ expr $now + 0.01 ] "finish"
178   } else {
179     $ns at [ expr $now + 1 ] "convergence-test"
180   }
181 }
182
183  proc dump-topology-graph {} {
184   global tma ns
```

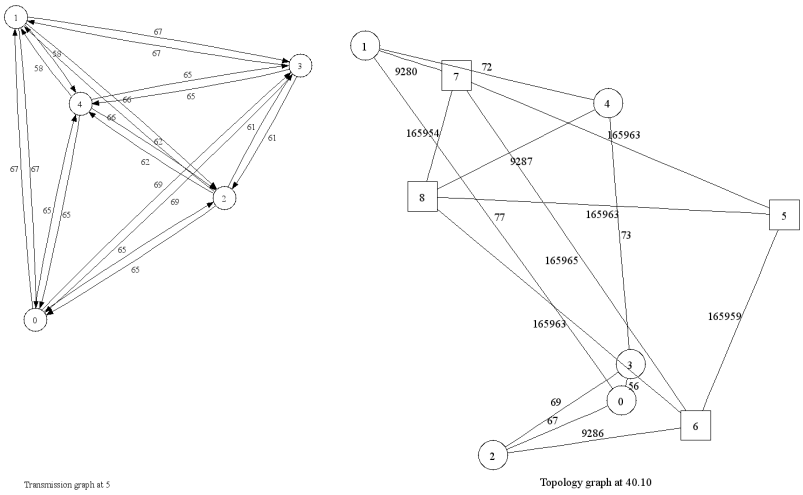
```
185   set now [ [ $ns set scheduler_ ] now ]
186   $tma(0) dump-topology-graph topology-graph-t[ expr round($now) ]
          -neato.dot
187 }
188
189 proc dump-topology-tree {} {
190   global tma ns
191   set now [ [ $ns set scheduler_ ] now ]
192   $tma(0) dump-topology-tree topology-tree-t[ expr round($now) ]-dot.dot
193 }
194
195 #-------------------------------------------------------------------------
196 # Create network topology
197 #-------------------------------------------------------------------------
198 source $topology_file
199
200 #-------------------------------------------------------------------------
201 # Get an instance to the TMA objects and obtain the MAC address of each
202 # node.
203 #-------------------------------------------------------------------------
204 for { set i 0 } { $i < [ array size node_ ] } { incr i } {
205   set tma($i) [ $node_($i) set tma_(0) ]
206   set mac($i) [ [ $node_($i) set mac_(0) ] id ]
207   $ns at 0.0  "$tma($i) start"
208 }
209
210 if { $topo_thallner_k != [ $tma(0) thallner-k ] } {
211   puts stderr "error: thallner k is different ($topo_thallner_k != [ $tma
          (0) thallner-k ])!"
212   exit 1
213 }
214
215 #-------------------------------------------------------------------------
216 # Create traffic patterns
217 #-------------------------------------------------------------------------
218
219
220 #-------------------------------------------------------------------------
221 # Time schedule
222 #-------------------------------------------------------------------------
223   $ns at 5.0 "convergence-test"
224 $ns run
```

C.4 Examples

C.4.1 Flooding Example with UDP/CBR Traffic

This example assumes a network with 5 nodes and 4 gateway nodes. The transmission graph is shown in Figure C.1(a) and the final network topology is shown in Figure C.1(b). In this example the network topology has converged after 10 seconds. Immediately after startup the UDP traffic between node 2 and node 1 is started using a packet size of 50

Bytes and an interval of $200ms$.



(a) Transmission Graph (b) Topology Graph

Figure C.1: Transmission and topology graph

What we can see from the final topology graph is that traffic from node 2 to node 1 can not be transmitted directly. Instead, alternate paths must be used, where possible ones are for example $2-0-1$, $2-3-0-1$, $2-3-4-8-1$, $2-3-4-8-1$, $2-6-5-8-7-1$, $2-6-5-8-4-1$. $0-3-2-1$ or $0-3-2-4-1$. Sending a packet over all possible paths is not very efficient and therefore the SSF protocol does not explore all paths. It forwards a packet only once and therefore reduces (or cuts) all possible paths during execution. An example output from the trace file is shown in Listing C.5.

In lines $1-3$ node 2 (-Ni 2) sends the UDP packet which is transmitted by the SSF protocol using the sequence number 196 (-Ps in line listing:example:udp-cbr:flooding:tracefile-n2-send-stop). In line 4, node 0 receives the packet and forwards it again in 5. The same holds for node 3, which receives it in line 6 and forwards it in 7. Node 6 receives the message in line 8 and forwards it in 9.
Now node 0 (-Ni 0) receives the previously forwarded packet from node 3 (-Ms 3) in line 10, but drops it because it has already forwarded it[1]. The same holds for node 2 shown in line 12. Node 4 first receives the packet it in line 14, and because it has not previously forwarded it, it does so in line 16. Lines $17-20$ show packets dropped which were forwarded by node 0. Finally, in line 21, node 1 receives the packet and passes it to the agent in line 22. The rest of the trace file has been stripped, because it only shows more forwarded and dropped packets.

[1]This is one of the most important features of our SSF protocol.

Listing C.5: Example output from tracefile for flooding protocol

```
1  s -t 39.100000000 -Hs 2 -Hd -2 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
       -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 2.0 -Id 1.0 -It
       cbr -Il 50 -If 0 -Ii 195 -Iv 32 -Pn cbr -Pi 195 -Pf 0 -Po 0
2  r -t 39.100000000 -Hs 2 -Hd -2 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 2.0 -Id 1.0 -It
       cbr -Il 50 -If 0 -Ii 195 -Iv 32 -Pn cbr -Pi 195 -Pf 0 -Po 0
3  s -t 39.100000000 -Hs 2 -Hd -1 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 2.0 -Id 1.0 -It
       cbr -Il 74 -If 0 -Ii 195 -Iv 31 -P SSF -Ps 196 -Pn cbr -Pi 195 -Pf 0 -
       Po 0
4  r -t 39.101128078 -Hs 0 -Hd -1 -Ni 0 -Nx 65.33 -Ny 26.31 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 2 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 31 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 1 -Po 0
5  f -t 39.101128078 -Hs 0 -Hd -1 -Ni 0 -Nx 65.33 -Ny 26.31 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 2 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 1 -Po 0
6  r -t 39.101128092 -Hs 3 -Hd -1 -Ni 3 -Nx 66.75 -Ny 32.32 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 2 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 31 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 1 -Po 0
7  f -t 39.101128092 -Hs 3 -Hd -1 -Ni 3 -Nx 66.75 -Ny 32.32 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 2 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 1 -Po 0
8  r -t 39.101128115 -Hs 6 -Hd -1 -Ni 6 -Nx 77.79 -Ny 22.12 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 2 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 31 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 1 -Po 0
9  f -t 39.101128115 -Hs 6 -Hd -1 -Ni 6 -Nx 77.79 -Ny 22.12 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 2 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 1 -Po 0
10 r -t 39.102376113 -Hs 0 -Hd -1 -Ni 0 -Nx 65.33 -Ny 26.31 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 3 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 2 -Po 0
11 d -t 39.102376113 -Hs 0 -Hd -1 -Ni 0 -Nx 65.33 -Ny 26.31 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw LOOP -Ma 0 -Md ffffffff -Ms 3 -Mt 800 -Is 2.0 -Id
        1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 29 -P SSF -Ps 196 -Pn cbr -Pi 195
        -Pf 2 -Po 0
12 r -t 39.102376184 -Hs 2 -Hd -1 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 3 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 2 -Po 0
13 d -t 39.102376184 -Hs 2 -Hd -1 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw LOOP -Ma 0 -Md ffffffff -Ms 3 -Mt 800 -Is 2.0 -Id
        1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 29 -P SSF -Ps 196 -Pn cbr -Pi 195
        -Pf 2 -Po 0
14 r -t 39.102376239 -Hs 4 -Hd -1 -Ni 4 -Nx 63.10 -Ny 76.18 -Nz 0.00 -Ne
       -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 3 -Mt 800 -Is 2.0 -Id
```

```
      1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
      -Pf 2 -Po 0
15 f -t 39.102376239 -Hs 4 -Hd -1 -Ni 4 -Nx 63.10 -Ny 76.18 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 3 -Mt 800 -Is 2.0 -Id
      1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 29 -P SSF -Ps 196 -Pn cbr -Pi 195
      -Pf 2 -Po 0
16
17 r -t 39.103534133 -Hs 3 -Hd -1 -Ni 3 -Nx 66.75 -Ny 32.32 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 0 -Mt 800 -Is 2.0 -Id
      1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
      -Pf 2 -Po 0
18 d -t 39.103534133 -Hs 3 -Hd -1 -Ni 3 -Nx 66.75 -Ny 32.32 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw LOOP -Ma 0 -Md ffffffff -Ms 0 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 29 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 2 -Po 0
19 r -t 39.103534191 -Hs 2 -Hd -1 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 0 -Mt 800 -Is 2.0 -Id
      1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
      -Pf 2 -Po 0
20 d -t 39.103534191 -Hs 2 -Hd -1 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw LOOP -Ma 0 -Md ffffffff -Ms 0 -Mt 800 -Is 2.0 -Id
       1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 29 -P SSF -Ps 196 -Pn cbr -Pi 195
       -Pf 2 -Po 0
21 r -t 39.103534357 -Hs 1 -Hd -1 -Ni 1 -Nx 22.31 -Ny 85.76 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw --- -Ma 0 -Md ffffffff -Ms 0 -Mt 800 -Is 2.0 -Id
      1.0 -It cbr -Il 74 -If 0 -Ii 195 -Iv 30 -P SSF -Ps 196 -Pn cbr -Pi 195
      -Pf 2 -Po 0
22 r -t 39.103534357 -Hs 1 -Hd -1 -Ni 1 -Nx 22.31 -Ny 85.76 -Nz 0.00 -Ne
      -1.000000 -Nl AGT -Nw --- -Ma 0 -Md ffffffff -Ms 0 -Mt 800 -Is 2.0 -Id
      1.0 -It cbr -Il 50 -If 0 -Ii 195 -Iv 29 -P SSF -Ps 196 -Pn cbr -Pi 195
      -Pf 2 -Po 0
23 ...
```

The script used for this simulation is shown in Listing C.6. The topology used is shown in C.7.

Listing C.6: NS2 simulator setup script for SSF/UDP-CBR example

```
1  #---------------------------------------------------------------------
2  # Project: Template for NS2 simulation framework
3  # Author:  Christian Walter <e0225458@student.tuwien.ac.at>
4  #
5  # $Log: usage-flooding-cbr-adhoc.tcl,v $
6  # Revision 1.3  2007/10/10 19:49:00  cwalter
7  #  - Final updates.
8  #
9  # Revision 1.2  2007/08/22 19:37:46  cwalter
10 #  - First corrected and updated version.
11 #
12 #---------------------------------------------------------------------
13
14 #---------------------------------------------------------------------
15 # set configuration properties
16 #---------------------------------------------------------------------
17 set topo_thallner_k      3
```

```
18 set num_nodes            100                          ;# total nodes
19 set xsize                100
20 set ysize                100
21
22 set val(chan)            Channel/WirelessChannel      ;# Channel Type
23 set val(prop)            Propagation/TwoRayGround     ;# radio-propagation
       model
24 set val(netif)           Phy/WirelessPhy              ;# network interface
       type
25 set val(mac)             Mac/802_11
26 set val(tma)             TMA/Thallner
27 set val(ifq)             Queue/DropTail/PriQueue      ;# interface queue type
28 set val(ll)              LL                           ;# link layer type
29 set val(ant)             Antenna/OmniAntenna          ;# antenna model
30 set val(ifqlen)          100                          ;# max packet in ifq
31 set val(rp)              SSF
32
33 # ======================================================================
34 # Configure for IEEE802.11b
35 # ======================================================================
36 Mac/802_11 set SlotTime_           0.000020          ;# 20us
37 Mac/802_11 set SIFS_               0.000010          ;# 10us
38 Mac/802_11 set PreambleLength_     144               ;# 144 bit
39 Mac/802_11 set PLCPHeaderLength_   48                ;# 48 bits
40 Mac/802_11 set PLCPDataRate_       1.0e6             ;# 1Mbps
41 Mac/802_11 set dataRate_           11.0e6            ;# 11Mbps
42 Mac/802_11 set basicRate_          1.0e6             ;# 1Mbps
43
44 Phy/WirelessPhy set freq_         2.4e+9            ;# Frequency
45 Phy/WirelessPhy set Pt_           3.3962527e-2      ;# Transmission power
46 Phy/WirelessPhy set RXThresh_     6.309573e-12      ;# Receiver threshold
47 Phy/WirelessPhy set CSThresh_     6.309573e-12      ;# Sense threshold
48
49 TMA/Thallner set debug_ false
50
51 # ======================================================================
52 # Create a simulator instance
53 # ======================================================================
54 set ns [ new Simulator ]
55 set tracefd [ open adhoc.tr w ]
56 $ns use-newtrace
57 $ns trace-all $tracefd
58
59 set topo [ new Topography ]
60 $topo load_flatgrid $xsize $ysize
61
62 set god_ [ create-god $num_nodes ]
63
64 set chan_1_ [new $val(chan)]
65 $ns node-config \
66      -adhocRouting $val(rp) \
67      -llType $val(ll) \
68      -macType $val(mac) \
69      -tmaType $val(tma) \
```

```
70      -phyType $val(netif) \
71      -ifqType $val(ifq) \
72      -ifqLen $val(ifqlen) \
73      -antType $val(ant) \
74      -propType $val(prop) \
75      -topoInstance $topo \
76      -agentTrace ON \
77      -routerTrace ON \
78      -macTrace OFF  \
79      -tmaTrace OFF \
80      -movementTrace OFF \
81      -channel $chan_1_
82
83  #-----------------------------------------------------------------
84  # Common functions
85  #-----------------------------------------------------------------
86  proc finish {} {
87      global ns tracefd
88      $ns flush-trace
89      close $tracefd
90      exit 0
91  }
92
93  proc convergence-calculate-times { arrname } {
94    global tma topo_thallner_k
95    upvar $arrname times
96
97    set num_nodes [ array size tma ]
98
99    # floor( (n-1)/(k-1) ) = ngroups is the number of groups. If the
100   # tree is balanced the height is log( ngroups ) / log( k ). Every
101   # node therefore walks up this hierarchy and creates group proposals
102   # where 1 is added because of the proposal for the node itself.
103   set group_tree_depth [ expr log( ( $num_nodes - 1 )/( $topo_thallner_k -
          1 ) ) / log( $topo_thallner_k ) + 1 ]
104
105   # add an extra factor of 2 because a generated proposal must also
106   # be accepted by the thallner algorithm.
107   set times(convergence_time_proposals) [ expr [ $tma(0) proposals-period ]
           * $group_tree_depth * 2.0 ]
108
109   # calculate the time required to check all groups at a node
110   set times(convergence_time_group) [ expr [ $tma(0) groupcheck-period ] *
          $group_tree_depth ]
111
112   # the convergence time is the maximum
113   if { $times(convergence_time_group) < $times(convergence_time_proposals)
          } {
114     set times(convergence_time) $times(convergence_time_proposals)
115   } else {
116     set times(convergence_time) $times(convergence_time_group)
117   }
118 }
119
```

```
120 proc convergence-test {} {
121   global ns tma val last_stats_time topo_prefix topo_thallner_k
122   set scheduler [ $ns set scheduler_ ]
123   set now [ $scheduler now ]
124
125   convergence-calculate-times times
126
127   # we assume that the network has converged.
128   set is_stable 1
129
130   # compute the time when the group structures have changed.
131   set last_change_max 0.0
132   for { set i 0 } { $i < [ array size tma ] } { incr i } {
133     set last_change [ $tma($i) last-change ]
134     if { $last_change > $last_change_max } {
135       set last_change_max $last_change
136     }
137     # if this node is not a gateway node it must have a node
138     # degree of k.
139     if { 0 == [ $tma($i) gateway-node ] } {
140       if { $topo_thallner_k != [ $tma($i) node-degree ] } {
141         set is_stable 0
142       }
143     }
144   }
145
146   # if the group structures have not changed during the convergence
147   # time the network is stable.
148   if { $now > $times(convergence_time) } {
149     if { $now - $last_change_max < $times(convergence_time) } {
150       puts "group structure has changed at time $last_change"
151       set is_stable 0
152     }
153   } else {
154     set is_stable 0
155   }
156
157   if { $is_stable == 1 } {
158     dump-topology-graph
159     dump-topology-tree
160     dump-transmission-graph
161     dump-ssf-stats
162     $ns at [ expr $now + 0.01 ] "finish"
163   } else {
164     $ns at [ expr $now + 1 ] "convergence-test"
165   }
166 }
167
168 proc dump-transmission-graph {} {
169   global tma ns
170   set now [ [ $ns set scheduler_ ] now ]
171   $tma(0) dump-transmission-graph transmission-graph-neato.dot
172 }
173
```

```
174 proc dump-topology-graph {} {
175   global tma ns
176   set now [ [ $ns set scheduler_ ] now ]
177   $tma(0) dump-topology-graph topology-graph-t[ expr round($now) ]
          -neato.dot
178 }
179
180 proc dump-topology-tree {} {
181   global tma ns
182   set now [ [ $ns set scheduler_ ] now ]
183   $tma(0) dump-topology-tree topology-tree-t[ expr round($now) ]-dot.dot
184 }
185
186 proc dump-ssf-stats {} {
187   foreach agent [ Agent/SSF info instances ] {
188     $agent statistics
189   }
190 }
191
192 #-------------------------------------------------------------------
193 # Create network topology
194 #-------------------------------------------------------------------
195 source adhoc-topology.tcl
196
197 #-------------------------------------------------------------------
198 # Get an instance to the TMA objects and obtain the MAC address of each
199 # node.
200 #-------------------------------------------------------------------
201 for { set i 0 } { $i < [ array size node_ ] } { incr i } {
202   set tma($i) [ $node_($i) set tma_(0) ]
203   set mac($i) [ [ $node_($i) set mac_(0) ] id ]
204   $ns at 0.0  "$tma($i) start"
205 }
206
207 #-------------------------------------------------------------------
208 # Create traffic patterns
209 #-------------------------------------------------------------------
210 # UDP traffic from node 2 to node 1 ( 50bytes, 200ms interval )
211 set udp0 [ new Agent/UDP ]
212 $ns attach-agent $node_(2) $udp0
213 set cbr0 [ new Application/Traffic/CBR ]
214 $cbr0 set packetSize_ 50B
215 $cbr0 set interval_   200ms
216 $cbr0 attach-agent $udp0
217 set loss0 [ new Agent/LossMonitor ]
218 $ns attach-agent $node_(1) $loss0
219 $ns connect $udp0 $loss0
220
221 #-------------------------------------------------------------------
222 # Time schedule
223 #-------------------------------------------------------------------
224 $ns at 0.1 "convergence-test"
225 $ns at 0.1 "$cbr0 start"
226 $ns run
```

Listing C.7: NS2 network topology script for SSF/UDP-CBR example

```
set node_(0) [ $ns node ]
$node_(0) set X_ 65.325537680334207
$node_(0) set Y_ 26.311793376836828

set node_(1) [ $ns node ]
$node_(1) set X_ 22.311284496593885
$node_(1) set Y_ 85.758534253462471

set node_(2) [ $ns node ]
$node_(2) set X_ 43.685197943674957
$node_(2) set Y_ 17.121839345023893

set node_(3) [ $ns node ]
$node_(3) set X_ 66.753871816561499
$node_(3) set Y_ 32.323620949091215

set node_(4) [ $ns node ]
$node_(4) set X_ 63.09729137602136
$node_(4) set Y_ 76.176156791009092

set node_(5) [ $ns node ]
$node_(5) set X_ 92.667186489639434
$node_(5) set Y_ 57.403331369815078
set tma_(5) [ $node_(5) set tma_(0) ]
$tma_(5) gateway-node 1

set node_(6) [ $ns node ]
$node_(6) set X_ 77.790332482098762
$node_(6) set Y_ 22.118026633801882
set tma_(6) [ $node_(6) set tma_(0) ]
$tma_(6) gateway-node 1

set node_(7) [ $ns node ]
$node_(7) set X_ 37.673634308238348
$node_(7) set Y_ 80.771818561838856
set tma_(7) [ $node_(7) set tma_(0) ]
$tma_(7) gateway-node 1

set node_(8) [ $ns node ]
$node_(8) set X_ 31.954568825640983
$node_(8) set Y_ 60.438252547959912
set tma_(8) [ $node_(8) set tma_(0) ]
$tma_(8) gateway-node 1
```

C.4.2 Routing Protocol with UDP/CBR Traffic

We assume the same network as shown in Figure C.1(a), with the final network topology shown in Figure C.1(b). Instead of a flooding protocol, we have decided to use the *Destination-Sequenced Distance-Vector* routing protocol. A complete description of DSDV is given in [PB94]. DSDV includes extensions to the classic Bellmann-Ford algorithm [Bel58] to improve the poor loop performance. The basic operation is like the

classic algorithm , but each route to a destination is tagged with a sequence number used to avoid the occurrence of routing loops. All sequence numbers originate at the destination, are by convention even numbers [PB94, 4], and are incremented prior to sending the routing updates [PB94, p.3]. Like in the classic Bellmann-Ford algorithm, each station periodically transmits updates. The simulation settings are the same as in the example shown in Section C.4.1.

What we can see in Figure C.1(b) is that a packet cannot be transmitted directly from node 2 to node 1. This becomes obvious if we look at the NS2 trace file, where the routing protocol forwards packets over multiple nodes. An example output is shown in the trace file in Listing C.8. In line 1, node 2 starts the transmission by passing the message to the routing agent, which receives it in line 2. The routing agent then forwards the packet to node 0 in line 3. Node 0 receives the packet in line 4, and because it is not the destination node, it forwards the packet to node 1 in line 5. Finally, node 1 accepts the packet in line 6.

Listing C.8: Example output from tracefile for DSDV routing protocol

```
1 s -t 100.000000000 -Hs 2 -Hd -2 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 2.0 -Id 1.0 -It
    cbr -Il 50 -If 0 -Ii 128 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
2 r -t 100.000000000 -Hs 2 -Hd -2 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
    -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 2.0 -Id 1.0 -It
    cbr -Il 50 -If 0 -Ii 128 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
3 s -t 100.000000000 -Hs 2 -Hd 0 -Ni 2 -Nx 43.69 -Ny 17.12 -Nz 0.00 -Ne
    -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 2.0 -Id 1.0 -It
    cbr -Il 70 -If 0 -Ii 128 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
4 r -t 100.003607549 -Hs 0 -Hd 0 -Ni 0 -Nx 65.33 -Ny 26.31 -Nz 0.00 -Ne
    -1.000000 -Nl RTR -Nw --- -Ma 13a -Md 0 -Ms 2 -Mt 800 -Is 2.0 -Id 1.0 -
    It cbr -Il 70 -If 0 -Ii 128 -Iv 32 -Pn cbr -Pi 0 -Pf 1 -Po 0
5 f -t 100.003607549 -Hs 0 -Hd 1 -Ni 0 -Nx 65.33 -Ny 26.31 -Nz 0.00 -Ne
    -1.000000 -Nl RTR -Nw --- -Ma 13a -Md 0 -Ms 2 -Mt 800 -Is 2.0 -Id 1.0 -
    It cbr -Il 70 -If 0 -Ii 128 -Iv 31 -Pn cbr -Pi 0 -Pf 1 -Po 0
6 r -t 100.007300261 -Hs 1 -Hd 1 -Ni 1 -Nx 22.31 -Ny 85.76 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 13a -Md 1 -Ms 0 -Mt 800 -Is 2.0 -Id 1.0 -
    It cbr -Il 70 -If 0 -Ii 128 -Iv 31 -Pn cbr -Pi 0 -Pf 2 -Po 0
```

The script used for this simulation is the same as in Listing C.6, with the exception that the routing protocol has been changed to DSDV. The modifications are shown in Listing C.9.

Listing C.9: Modifications for NS2 setup script for DSDV/UDP-CBR example

```
1  ...
2  # ======================================================================
3  # set configuration properties
4  # ======================================================================
5  set num_nodes               100                    ;# total nodes
6
7  ...
8  set val(ifqlen)             100                    ;# max packet in ifq
9  set val(rp)                 DSDV
10
```

```
11 set val(tma_cotime)      20                          ;# convergence time
12 ...
```

C.4.3 Example for TMA/Filter with UDP/CBR traffic and DSDV

This example shows the usage of the TMA/Filter module. In the setup script shown in Listing C.10, we create three wireless nodes 0, 1, and 2, and limit node 0's communication to node 1 and node 2's communication to node 1. Node 1s communication is not restricted at all. Furthermore, we create a UDP traffic instance between node 0 and node 2. The DSDV routing protocol will be used to establish routing paths between the nodes.
After the routing protocol has converged, node 0 will send packets to node 1 directly, and to node 2 using the gateway node 1 because the direct communication has been restricted by the TMA. This can be seen by looking at the trace file output, which is shown in Listing C.11, where a packet is sent from node 0 to node 2. Note the forwarding of the packet in line 6 by node 1.

Listing C.10: NS2 network topology script for TMA/Filter and DSDV

```
1  #-----------------------------------------------------------------------
2  # Project: Example for the TMA/Filter module.
3  # Author:  Christian Walter <e0225458@student.tuwien.ac.at>
4  #
5  # $Log: usage-tma-filter-cbr-dsdv-adhoc.tcl,v $
6  # Revision 1.2  2007/10/10 19:49:00  cwalter
7  #  - Final updates.
8  #
9  # Revision 1.1  2007/07/29 21:36:33  cwalter
10 #  - New code examples for ns2 chapter.
11 #
12 # Revision 1.2  2007/07/28 14:13:20  cwalter
13 #  - Added debug flag.
14 #
15 # Revision 1.1  2007/07/28 14:04:15  cwalter
16 #  - Added example script for TMA/Filter module.
17 #
18 #
19 #-----------------------------------------------------------------------
20
21 #-----------------------------------------------------------------------
22 # set configuration properties
23 #-----------------------------------------------------------------------
24 set num_nodes               3                           ;# total nodes
25 set xsize                   100
26 set ysize                   100
27
28 set val(chan)               Channel/WirelessChannel     ;# Channel Type
29 set val(prop)               Propagation/TwoRayGround    ;# radio-propagation
       model
30 set val(netif)              Phy/WirelessPhy             ;# network interface
       type
```

```
31 set val(mac)             Mac/802_11
32 set val(tma)             TMA/Filter
33 set val(ifq)             Queue/DropTail/PriQueue      ;# interface queue type
34 set val(ll)              LL                           ;# link layer type
35 set val(ant)             Antenna/OmniAntenna          ;# antenna model
36 set val(ifqlen)          100                          ;# max packet in ifq
37 set val(rp)              DSDV
38
39 #TMA/Filter set debug_ true
40
41 # ======================================================================
42 # Create a simulator instance
43 # ======================================================================
44 set ns [ new Simulator ]
45 set tracefd [ open adhoc.tr w ]
46 $ns use-newtrace
47 $ns trace-all $tracefd
48
49 set topo [ new Topography ]
50 $topo load_flatgrid $xsize $ysize
51
52 set god_ [ create-god $num_nodes ]
53
54 set chan_1_ [new $val(chan)]
55 $ns node-config \
56      -adhocRouting $val(rp) \
57      -llType $val(ll) \
58      -macType $val(mac) \
59      -tmaType $val(tma) \
60      -phyType $val(netif) \
61      -ifqType $val(ifq) \
62      -ifqLen $val(ifqlen) \
63      -antType $val(ant) \
64      -propType $val(prop) \
65      -topoInstance $topo \
66      -agentTrace ON \
67      -routerTrace ON \
68      -macTrace OFF \
69      -tmaTrace OFF \
70      -movementTrace OFF \
71      -channel $chan_1_
72
73 # ======================================================================
74 # Common functions
75 # ======================================================================
76 proc finish {} {
77     global ns tracefd
78     $ns flush-trace
79     close $tracefd
80     exit 0
81 }
82
83 # ======================================================================
84 # Create nodes
```

```
85 #---------------------------------------------------------------------------
86 for { set i 0 } { $i < $num_nodes } { incr i } {
87   set node_($i) [ $ns node ];
88   $node_($i) random-motion 0
89   set tma($i) [ $node_($i) set tma_(0) ]
90   set mac($i) [ [ $node_($i) set mac_(0) ] id ];
91 }
92
93 $node_(0) set X_ 10.0; $node_(0) set Y_ 15.0; $node_(0) set Z_ 0.0;
94 $node_(1) set X_ 60.0; $node_(1) set Y_ 75.0; $node_(1) set Z_ 0.0;
95 $node_(2) set X_ 30.0; $node_(2) set Y_ 40.0; $node_(2) set Z_ 0.0;
96
97 #---------------------------------------------------------------------------
98 # allow given nodes to talk to each other.
99 #---------------------------------------------------------------------------
100 $tma(0) add-neighbor $mac(1)
101 $tma(1) add-neighbor $mac(0) $mac(2)
102 $tma(2) add-neighbor $mac(1)
103
104 #---------------------------------------------------------------------------
105 # Create traffic patterns
106 #---------------------------------------------------------------------------
107
108 # UDP traffic from node 0 to node 2
109 set udp0 [ new Agent/UDP ]
110 $ns attach-agent $node_(0) $udp0
111 set cbr0 [ new Application/Traffic/CBR ]
112 $cbr0 set packetSize_ 50B
113 $cbr0 set interval_   200ms
114 $cbr0 attach-agent $udp0
115 set loss0 [ new Agent/LossMonitor ]
116 $ns attach-agent $node_(2) $loss0
117 $ns connect $udp0 $loss0
118
119 #---------------------------------------------------------------------------
120 # Time schedule
121 #---------------------------------------------------------------------------
122 $ns at 100.0  "$cbr0_start"
123 $ns at 101.0  "$cbr0_stop"
124 $ns at 300.0 "finish"
125 $ns run
```

Listing C.11: Example output from tracefile for DSDV and TMA/Filter module

```
1 ...
2 s -t 100.000000000 -Hs 0 -Hd -2 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -Ne
    -1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 2.0 -It
    cbr -Il 50 -If 0 -Ii 30 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
3 r -t 100.000000000 -Hs 0 -Hd -2 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -Ne
    -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 2.0 -It
    cbr -Il 50 -If 0 -Ii 30 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
4 s -t 100.000000000 -Hs 0 -Hd 1 -Ni 0 -Nx 10.00 -Ny 15.00 -Nz 0.00 -Ne
    -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 2.0 -It
    cbr -Il 70 -If 0 -Ii 30 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
```

```
5 r -t 100.004468822 -Hs 1 -Hd 1 -Ni 1 -Nx 60.00 -Ny 75.00 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw --- -Ma 13a -Md 1 -Ms 0 -Mt 800 -Is 0.0 -Id 2.0 -
      It cbr -Il 70 -If 0 -Ii 30 -Iv 32 -Pn cbr -Pi 0 -Pf 1 -Po 0
6 f -t 100.004468822 -Hs 1 -Hd 2 -Ni 1 -Nx 60.00 -Ny 75.00 -Nz 0.00 -Ne
      -1.000000 -Nl RTR -Nw --- -Ma 13a -Md 1 -Ms 0 -Mt 800 -Is 0.0 -Id 2.0 -
      It cbr -Il 70 -If 0 -Ii 30 -Iv 31 -Pn cbr -Pi 0 -Pf 1 -Po 0
7 r -t 100.009800898 -Hs 2 -Hd 2 -Ni 2 -Nx 30.00 -Ny 40.00 -Nz 0.00 -Ne
      -1.000000 -Nl AGT -Nw --- -Ma 13a -Md 2 -Ms 1 -Mt 800 -Is 0.0 -Id 2.0 -
      It cbr -Il 70 -If 0 -Ii 30 -Iv 31 -Pn cbr -Pi 0 -Pf 2 -Po 0
8 ...
```

D SSF - Source Sequenced Flooding

The SSF protocol is an implementation of a flooding algorithm for the NS2 wireless model. Every packet sent in the network is extended by an additional header of 4 bytes. This header includes a sequence number, which is generated by the source whenever a packet is sent. The sequence number is a strictly monotonic increasing sequence, where the first valid sequence number is 1[1].
Every time a node receives a packet, it checks if the sequence number within the packet header is smaller than its last known sequence number from that source. If this is the case, the packet is dropped because the node has already processed it once. Otherwise, it updates its own sequence counter for that node, and either delivers the packet locally if the destination matches the node address, or forwards the packet by performing a MAC broadcast.

D.1 Usage

The SSF protocol can be enabled by the standard node configuration interface in NS2. To enable SSF at a specific node, the code in Listing D.1 is sufficient:

Listing D.1: Enable SSF routing at a wireless node

```
1 set val(rp)                SSF
2 ...
3 $ns node-config \
4      -adHocRouting $val(rp) \
5      -agentTrace OFF
6      ...
```

To enable the recording of additional trace information, the value of −agentTrace should be set to ON during node configuration. In addition, the SSF routing module supports some runtime statistics. The amount of information available depends on the compile time directive **SSF_EXTENDED_STATS** defined in `ssf/ssf.h`. The information from the routing agents can be obtained by the TCL Listing D.2:

Listing D.2: Getting Agent/SSF status information

```
1 proc stats {} {
2     foreach agent [ Agent/SSF info instances ] {
3         $agent statistics
4     }
5 }
```

[1] Additional precautions must be taken because integer numbers are finite. Because we were only interested in a very simple test of flooding protocols we have not implemented this which limits the number of packets which can be sent by a node to 2^32.

To enable debugging of the SSF protocol, the class variable debug_ in Agent/SSF should be set to true with Agent/SSF **set** debug_ true.

D.2 Implementation overview

In this section, the general structure of a routing agent is explained. Most of the information has been obtained from the NS2 manual [FV06, p143-146] and the NS2 source. The basic picture of a wireless node is shown in Figure D.1. An incoming packet is

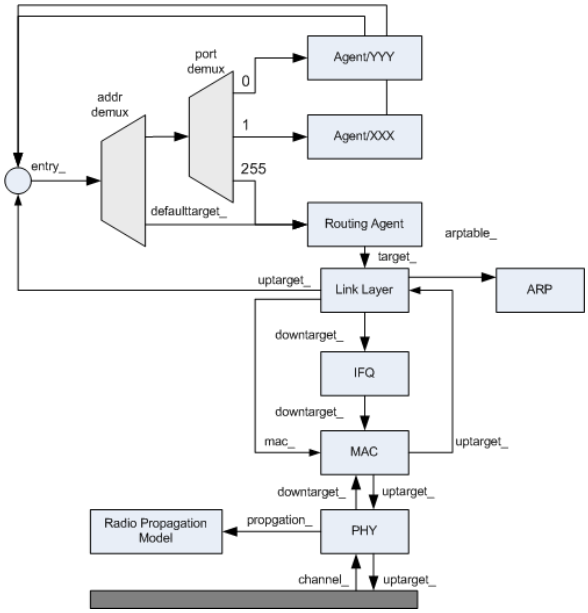


Figure D.1: Basic structure of a wireless node in NS2

passed from the MAC layer, which is defined in `mac/mac-802_11.{cc,h}`[2], to the link layer defined in `mac/ll.{cc,h}`. The link layer then passes the packet to the node entry point, which is by default the destination address classifier[3].
If the IP destination address matches the current node's address, the packet is passed to the port classifier, where the "normal" agents are attached. A normal agent is for example the Agent/LossMonitor. If the destination address does not match, the address classifier uses its defaulttarget_ and passes the packet to the routing agent. The routing agent then checks its routing table and decides whether to forward or drop the packet.
The case left is when a node sends a packet. In this case, the Agent creates a packet and again sends the packet to the node entry point. Normally, the packet will have a destination different from the source and the packet is passed to the routing agent to get forwarded.

[2]Assuming that Mac/802_11 is used.
[3]See the entry method defined in `tcl/lib/ns-node.tcl`.

The SSF routing agent[4] in NS2 works a bit differently. The reason is that a packet should always be passed to the SSF agent regardless of whether it is for the local host or another host. In Figure D.1, a packet received by a node for which it is designated would not be seen by the routing agent, because the destination address classifier would immediately handle it to the port demultiplexer. For this reason, SSF has a structure as shown in Figure D.2. Now every packet is first passed to the SSF Agent, which

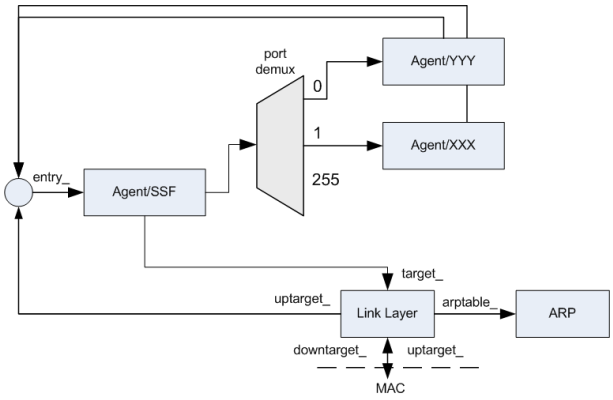


Figure D.2: Structure of a SSF wireless node in NS2

makes it possible to check the sequence number. If the Agent receives a packet for the local host, it checks its sequence number, and if it has not already seen it, it passes it to the local port demultiplexer. A packet is forwarded if its sequence number has not been seen. Because the structure of the node is different, it required some modifications in the node creation methods in `tcl/lib/ns-mobilenode.tcl` where the TCL class Node/MobileNode/Entry has been added. In addition, the node creation method create−wireless−node in `tcl/lib/ns-lib.tcl` has been modified to add the new routing agent.

D.3 Header format

The SSF protocol simply adds a sequence number to a normal Ethernet frame. The frame is shown in Table D.1. This implementation is somewhat theoretical, and a real world implementation would want to change the protocol type in the Ethernet frame and store the old one in the SSF header. This would allow compatibility with already existing protocols. For this work, however, our solution suffices.

[4]Similarly to the DSR routing agent.

Table D.1: SSF frame

MAC header				Frame Body		CRC
...	*Addr1*	*Addr2*	...	*Sequence*	*Original data*	...
...	`OR`	`FF`	...	$1 - 4294967296$	...	...

OR ... `OR:OR:OR:OR:OR:OR` is the source MAC address.
FF ... `FF:FF:FF:FF:FF:FF` is the MAC broadcast address.
Sequence ... The SSF sequence number.
Original data ... The original payload for the ethernet frame.

E Trace File Format

E.1 NS2 trace formats

The information collected here is based on the online documentation found in [ud06] and the NS2 manual [FV06]. The main element of a wireless trace file is shown in Table E.1. The value for the destination node (-Hd) may also be -1 or -2. In case of -1 the packet

Table E.1: NS2New wireless trace format

Event	*Abbreviation*	*Flag*	*Type*	*Value*
Wireless Event		-t	double	Time
		-Ni	int	Node ID
		-Nx	double	Node X Coordinate
		-Ny	double	Node Y Coordinate
		-Ne	double	Node Energy Level
	s: Send	-Ni	string	Network trace level (AGT, RTR, MAC, $\cdots$)
	r: Receive	-Nw	string	Drop reason
	d: Drop	-Hs	int	Hop source node ID
	f: Forward	-Hd	int	Hop destination ID, -1, -2
		-Ma	hexadecimal	Duration
		-Ms	hexadecimal	Source Ethernet Address
		-Md	hexadecimal	Destination Ethernet Address
		-Mt	hexadecimal	Ethernet type
		-P	hexadecimal	Packet type (arp, dsr, imep, tora, $\cdots$)
		-Pn	string	Packet type (cbr, tcp)

is a broadcast packet. -2 means that the destination node has not been set. Depending on the packet type, the following additional flags shown in Table E.2 are available and written to the trace file output.

Table E.2: Packet type dependent options

Event	*Flag*	*Type*	*Value*
ARP Trace	-Po	string	Request or reply
	-Pms	int	Source MAC address
	-Ps	int	Source address
	-Pmd	int	Destination MAC address

Event	*Flag*	*Type*	*Value*
	-Pd	int	Destination address
IP Trace	-Is	int.int	Source address and port
	-Id	int.int	Destination address and Port
	-It	string	Packet type
	-Il	int	Packet size
	-If	int	Flow ID
	-Ii	int	Unique ID
	-Iv	int	TTL
CBR Trace	-Pi	int	Sequence number
	-Pf	int	Number of times packet was forwarded
	-Po	int	Optimal number of forwards
SSF Trace	-Ps	int	Packet sequence number

List of Figures

List of Tables

Listings

Bibliography

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, March 2004.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[CBD02] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications & Mobile Computing (WCMC): Special Issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5):483–502, 2002.

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

[FV06] K. Fall and K. Varadhan. *The ns Manual,* *http://www.isi.edu/nsnam/ns/ns-documentation.html*. ONLINE, 2006.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[JMB01] D. Johnson, D. Maltz, and J. Broch. *DSR The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[JMC+01] Philippe Jacquet, Paul Mhlethaler, Thomas Clausen, Anis Laouiti, Amir Qayyum, and Laurent Viennot. Optimized link state routing protocol. In *IEEE INMIC'01, 28-30 December 2001, Lahore, Pakistan*, pages 62–68. IEEE, December 2001.

[JMH03] D. Johnson, D. Maltz, and Y Hu. The dynamic source routing protocol for mobile ad hoc networks (dsr), `http://www.cs.cmu.edu/~dmaltz/internet-drafts/draft-ietf-manet-dsr-09.txt`, 2003.

[Lan03] Daniel Lang. A comprehensive overview about selected ad hoc networking routing protocols. Technical report, Technische Universität München, Department of Computer Science, March 2003.

[LGLA98] Brian Neil Levine and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems*, 6(5):334–348, 1998.

[LHB+01] Li Li, Joseph Y. Halpern, Paramvir Bahl, Yi-Min Wang, and Roger Wattenhofer. Analysis of a cone-based distributed topology control algorithm for wireless multi-hop networks. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 264–273, New York, NY, USA, 2001. ACM Press.

[LHSS05] N. Li, J. C. Hou, C. Sha, and L. Sha. Design and analysis of an mst-based topology control algorithm. *IEEE Transaction on Wireless Communicaons*, 4(3), 2005.

[MGLA96] Shree Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *ACM/Baltzer Journal on Mobile Networks and Applications, Special Issue on Routing in Mobile Communication Networks*, 1(2):183–197, October 1996.

[Moy97] John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1997.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *SIGCOMM '94: Proceedings of the Conference on Communications Architectures, Protocols and Applications*, pages 234–244, New York, NY, USA, 1994. ACM Press.

[PC97] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *IEEE Conference on Computer Communications, INFOCOM'97, April 7-11, 1997, Kobe, Japan*, volume 3, pages 1405–1413. IEEE, April 1997.

[PDDJ02] V. Paruchuri, A. Durresi, D. Dash, and R. Jain. Optimal flooding protocol for routing in ad-hoc networks, 2002.

[PH06] Laurent Paquereau and Bjarne E. Helvik. A module-based wireless node for ns-2. In *WNS2 '06: Proceeding from the 2006 Workshop on ns-2: The IP network simulator*, page 4, New York, NY, USA, 2006. ACM Press.

[PR99] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. *WMCSA - Workshop on Mobile Computing Systems and Applications*, 1999.

[Raj02] R. Rajaraman. Topology control and routing in ad hoc networks: a survey, 2002. R. Rajaraman. Topology control and routing in ad hoc networks: a survey. SIGACT News, 2002.

[Ray96] Michel Raynal. Fault-tolerant distributed systems: a modular approach to the non-blocking atomic commitment problem. Technical Report RR-2973, IRISA, Campus de Beaulieu, 1996.

[RR04] Francisco J. Ros and Pedro M. Ruiz. Implementing a new manet unicast routing protocol in ns2, 2004. `http://masimum.dif.um.es/nsrt-howto/html/`.

[RT99] E. Royer and C. Toh. A review of current routing protocols for ad-hoc mobile wireless networks, 1999. E.M. Royer and C-K Toh. A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks. IEEE Personal Communications, Apr. 1999.

[San05] Paolo Santi. Topology control in wireless ad hoc and sensor networks. *ACM Computing Survey*, 37(2):164–194, 2005.

[Sch00] Jochen Schiller. *Mobile communications*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[SKSS04] Ghosh Sukumar, Lillis Kevin, Pandit Saurav, and Pemmaraju Sriram. Robust topology control algorithms, 2004. PG. Sukumar, L. Kevin, P. Saurav, P. Sriram. Robust topology control protocols. In OPODIS - International Conference on Principles of Distributed Systems. December 2004.

[Tha05] Bernd Thallner. Topology control for fault-tolerant communication in wireless ad hoc networks, PhD Thesis, 2005.

[THB+02] Jing Tian, Joerg Haehner, Christian Becker, Illya Stepanov, and Kurt Rothermel. Graph-based mobility model for mobile ad hoc network simulation. *ss*, 2002.

[TM05] Bernd Thallner and Heinrich Moser. Topology control for fault-tolerant communication in highly dynamic wireless networks, Dissertation. *The Third International Workshop on Intelligent Solutions in Embedded Systems (WISES 2005)*, May 2005.

[ud06] NS2 users and developers. The ns manual - online wiki, `http://nsnam.isi.edu/nsnam/index.php/Main_Page`, 2006.

[WZ04] Roger Wattenhofer and Aaron Zollinger. XTC: A Practical Topology Control Algorithm for Ad-Hoc Networks. In *4th International Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN), Santa Fe, New Mexico*, April 2004.