AG VERNETZTE SYSTEME
FACHBEREICH INFORMATIK
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Diplomarbeit

Cluster Based Routing in Mobile Ad Hoc Networks

Christoph Heidinger

---

2. Juni 2008

Ich erkläre hiermit, die vorliegende Diplomarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 2. Juni 2008

(Christoph Heidinger)

Cluster Based Routing in Mobile Ad Hoc Networks

Diplomarbeit

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Christoph Heidinger

Tag der Abgabe : 2. Juni 2008

Betreuer : Prof. Dr. Reinhard Gotzhein
Dipl. Inform. Alexander Geraldy

Contents

Chapter 1

Introduction

1.1 Background

This thesis presents the redesign and specification of a clustering algorithm for mobile ad hoc networks. *ReBaC2* is the improved version of the repair based clustering algorithm presented in [Hei07]. Based on ReBaC2, a composed routing scheme for mobile wireless ad hoc networks is also introduced.

A *wireless* network is a network of devices that communicate wirelessly with each other, e. g. most commonly over radio data transmission. The devices of the network are commonly called *nodes*. Such a network is termed *mobile* if the nodes have the possibility to change their location, possibly even while communicating with other nodes. An *ad hoc* network is a network without constantly available infrastructure, i. e. the network consists only of the nodes that are taking part, and no additional centrally controlling instances are present.

When mobile nodes wish to form a wireless ad hoc network, specific problems arise from the lack of infrastructure. A naive solution in this situation would be to use a leader election scheme in order to establish a central point of management. However, in networks with a large number of nodes, it is not practical to concentrate information on the complete network in a single node. Additional problems with this solution arise from the fact that the nodes may be constantly mobile while participating in the network, resulting in a highly dynamic network topology. Thus, the main problems in an ad hoc network have to be solved in a distributed manner.

One of the main problems to be solved for mobile ad hoc networks is *routing*. Routing is necessary in order to deliver data from one node of the network to another node of the network when the two nodes are not within direct reach

of each other. Routing normally involves the determination of paths for data packets to reach their destination from their source. There are several different approaches to routing in mobile ad hoc networks, some of which are presented in Section 1.2.

Routing in mobile ad hoc networks is generally difficult because in the beginning, the nodes need to distributedly establish an (incomplete) view of the network they are in, and after that they have to make routing decisions based on that incomplete knowledge. Additionally, with the mobility of the nodes, knowledge about network topology is reliable only for a short time.

Other challenges generally inherent to mobile wireless networks are the limited data transmission bandwidth and the limited energy availability at the mobile nodes. Both limitations demand an economical use of information transmission, especially concerning management messages.

In this work, several enhancements to ReBaC, the repair based clustering algorithm as presented in [Hei07], are devised. The resulting second version of the repair based clustering algorithm, ReBaC2, is used as a component in the composition of a hierarchical routing scheme for mobile ad hoc networks. The functionality of the composed routing scheme is demonstrated in simulations, and some further improvements are outlined.

1.2 Survey of Routing Schemes

In this section, a small selection of routing protocols for mobile ad hoc networks is presented. When considering routing protocols, there are some generic categories into which the routing protocols can be classified.

In *proactive* routing schemes, routing information, i. e. information on the network structure, is gathered before it is needed. This means that the algorithms start establishing the routing information in the nodes independently of whether a route is needed. On the other hand, *reactive* routing schemes start the route discovery only when a route is needed. This means that there is no control traffic when no routes are needed, but the establishment of a route starts only when it is actually needed for relaying a message. There is a case for a *hybrid* routing scheme, i. e. a combination of both approaches. An example is to use a proactive scheme within a certain region, and a reactive scheme outside of that region. The advantage of this combination is that the routes within the region are immediately available when needed, and there is less control traffic outside that region when fewer routes are needed outside the area.

Another differentiation of routing protocols is one between flat-routed and hierarchical schemes. In a *flat-routed* routing scheme, all nodes are equal, even after the construction of a routing structure. This means that every node has the same tasks and responsibilities, usually defined within its surroundings. With a *hierarchical* routing scheme, a hierarchy of nodes is established. This means that several of the nodes may be elected or otherwise chosen to perform additional activities that other nodes may make use of. This is usually done in order to reduce the network traffic by grouping several nodes together into organisational units that are self managed or managed by a specific node of the group.

1.2.1 AODV

The Ad-hoc On-Demand Distance Vector Routing scheme (AODV) as presented in [PR99] is a purely reactive routing scheme in that routes are only established when needed, and route information is only kept at nodes as long as required. When a route is needed and not known, the sender of the packet starts the route discovery process with a RREQ (route request) message. This message is flooded across the network until it reaches the destination provided by the original sender of the RREQ message. The intermediate nodes keep the so-called *reverse path* information towards the sender of the route request. This means that they record the node where the query originated, and the node the message was received from as the next hop towards the query's origin. On receipt of the RREQ message at the destination, a RREP (route reply) message is sent in reply along the recorded reverse path, establishing the *forward path* from source to destination. This is similar to the reverse path establishment in that the RREP message's origin and its last hop are recorded as local next-hop routing information. After the process of forward path establishment, a bidirectional path between source and destination has been set up. Note that the path is only present as single next hop information at intermediate nodes and there is no need to store the complete recorded path in the messages or in any node. Unused path information, such as the additionally recorded reverse path information at nodes not between source and destination, expires after some time.

In more detail, every node has two local counters, *node sequence number* and *broadcast ID*. The broadcast ID is used to implement a flooding algorithm for RREQ messages. The pair of node ID and broadcast ID is unique to every RREQ message that is sent. This makes sure that every RREQ message is only forwarded once by every node. A node that receives a new RREQ message rebroadcasts it and keeps the pair of its source node ID and broadcast

ID. When the same RREQ message arrives again via a different path, it is discarded and not rebroadcast.

The node sequence number of a node is used in routes towards that node as a degree of freshness of the route. Whenever a source starts a new route discovery, it includes its own node sequence number as well as the latest known sequence number of the destination in the RREQ message to the source. This makes sure that potential old route fragments are not considered valid for the new search anymore. The RREQ message is flooded through the network until it reaches either the destination itself or a node that has a route towards the destination that has a destination sequence number for that destination that is at least as high as the one in the RREQ message. At this node, the RREP message is generated, and the actual destination sequence number of the route that has been found is included in the RREP message. If the RREP message has reached the destination itself, and the destination sequence number of the request is higher than the current sequence number at the destination, the destination node increments its sequence number to the number received with the request. Based on the presence of destination sequence numbers in all routing entries, it is proved in [PR99] that the formation of routing loops is impossible, even in the face of high mobility.

There are several different timeout mechanisms. The *route request expiration timeout* determines the time for which the reverse path information is kept at a node when no RREP message is encountered by the node. After this time, it is reckoned that the RREP message has used another path or that the destination has not been found, and the reverse path information for this request is discarded.

The *route caching timeout* determines the time for which an inactive route is preserved in the routing table. Whenever a packet is transmitted over the route, the route caching timer for this route is reset. When the route has been unused for the given amount of time, it is considered that the route is not required anymore, and the routing information for this route is discarded. As the value for the route caching timeout is equal for all nodes, the route will be discarded at approximately the same time in all nodes that are part of a single path using the route. Note that one route towards a destination may be in use by several connections or other packet sequences between source and destination, at a certain node. The timeout mechanism means that the route is only discarded when it is not used by any packets anymore.

There is a timer called *active timeout* for each combination of direct neighbour and path destination. It determines the time for which a neighbour is considered active for a destination. The set of active nodes for a destination is used when the route to the destination breaks, and the information

on route breakage has to be relayed up towards the sources. On receiving the information of a broken route, either by locally detecting route breakage or by receiving a notification from nodes down the path, a node notifies all active neighbours for this destination of the incident. A route table entry is considered active if there is at least one active neighbour for its destination.

Route loss is detected with a combination of *hello interval* and the value for *allowed hello loss*. If a node did not send any messages within the last hello interval, it locally broadcasts a hello message in order to maintain the local connectivity information at its direct neighbours. If *allowed hello loss* hello messages have not been received from a neighbour, and no other messages have been received from that neighbour, it is considered lost. Another possibility of detecting the loss of a neighboring node are *link-layer acknowledgements*. This requires the data link layer to detect whether a frame that is sent is actually received by its destination, and to offer this information to the layer above. When a frame is sent without a link-level acknowledgement being received, the destination of the respective message is considered lost.

If the lost neighbour is the next hop of an active routing table entry, the respective route is considered broken and the active neighbours using the route are notified. This is done with a special unsolicited RREP message with infinite hop count. This notification message is relayed upwards until it reaches the source. The source can reinitiate a route discovery with an increased destination sequence number.

1.2.2 AODVlight

AODVlight is a simplification of AODV as described in Section 1.2.1. AODVlight has been designed to be an extremely light-weight routing protocol for use in mobile ad hoc networks that exhibits the main characteristics of AODV while at the same time being small in its specification size. It has been used as one of the routing schemes being combined in [FGG07].

AODVlight is based on AODV, but lacks some of the features of AODV. For example, in AODVlight, routes are only discovered unidirectionally. When a RREQ message is locally broadcast, the RREP message that is returned is sent back towards the source, but no routing information towards the source node is added to the routing table. In case the route in the opposite direction is also required, it needs to be established with a separate RREQ message.

Another limitation of AODVlight as compared to AODV is that there is no route caching timer keeping track of the usage of a route entry, and there is no route error detection in cases where the next hop of a route becomes

unavailable. Instead, there is a simple timeout mechanism that removes a route at a fixed time after its establishment. When a route is not in use anymore, nothing else is done. When the route is still required, it has to be re-established with a new RREQ message. In case of a route error, the route becomes unavailable due to the error until it is removed by the timeout mechanism. After removal, it is re-established by the first packet the source sends towards the destination.

1.2.3 DSDV

The Destination-Sequenced Distance-Vector routing as proposed in [PB94] is a proactive, flat-routed, multi-hop distance vector routing scheme that is proved to be loop-free at all times. The routing tables at every node contain an entry for every node in the complete network, and the nodes regularly broadcast their routing table to all direct neighbours. In order to facilitate a quick dissemination of current information, the contents of the routing tables are also broadcast when there has been significant change in the routing table. The routing table entries consist of the next hop node, the number of hops to the destination, and the destination sequence number.

Each node, when sending its routing table, in doing so announces its presence in the network. In order to make sure that only the freshest routes are used, the node's destination sequence number is incremented each time it sends its routing table. (The sequence numbers generated at the node itself are always even.) When a routing table broadcast is received, the receiving node adds or updates the route to the sender node, and also updates the routes to other nodes known by the sender if they are better. A route is considered better either if its sequence number is higher or if the sequence number is equal and the number of hops is smaller.

The loss of a connection is either detected by the data link layer when the connection is used, or it is inferred from the absence of regular routing table broadcasts. If the link to a direct neighbour has been lost, all routes that use this neighbour as next hop are assigned a hop count of infinity and their sequence number is incremented by one. This means that a sequence number generated at a node that is different from the actual destination node will always be odd, and thus the next sequence number generated by the destination itself will always be greater than the one generated at the node where the breakage of the route was detected.

There are some problems that are also mentioned in [PB94]. The first problem is that the regularly broadcast routing tables gain considerable size and

thus cause a significant network load in a network with a high total number of nodes. The suggested solution is that the nodes do not send their complete routing table every time, but that incremental routing table update messages are used that only contain the routing table entries that have changed in comparison to the previously announced state. Of course, due to the mobility of the nodes, full routing table messages have to be sent regularly, after longer intervals of time, so that nodes coming into the neighbourhood of another node have the possibility to eventually receive the full routing table information.

The second problem that is addressed in [PB94] is that it may be the case that a node regularly receives newer, but worse, routing information to a certain destination earlier than information on a better route that is also continuously present. If not addressed, this effect may lead to the affected node regularly broadcasting routing table changes, thus introducing fluctuation into the complete network even if there is no node mobility. The suggested solution proposes a delay in the propagation of the newer routing information if better information can be expected after a short while. The implementation of this additional feature introduces a second routing table in order to separate the routes that are broadcast to the neighbours from the ones actually in use by the node that will only later be transferred to the table that is broadcast.

1.2.4 ZRP

The Zone Routing Protocol as described in [Haa97] is an example of a hybrid routing scheme that combines proactive and reactive functionality. It constructs a flat-routed network structure.

The *zone* of each node is defined to consist of all nodes of the network that lie within a radius of r_{zone} hops from the node. Thus, the zones are heavily overlapping. Within a zone, a standard proactive routing scheme can be used to establish the routes, and thus a node knows all other nodes that lie within its cluster.

Between the zones, a reactive routing scheme is devised. Each node queries the outermost nodes of its zone, which in turn will query their outermost nodes, until the destination has been found. This is equivalent to flooding, except that only the outermost nodes of each zone take part in the process. This results in a lower number of nodes being involved in the information dissemination process, and far fewer messages being sent, because the messages are delivered directly, i. e. as unicast messages, towards the outermost

nodes of the zone. The routing paths resulting from this technique consist of a sequence of nodes each at the outer edge of the last node's zone, thus approximately r_{zone} away from each other. This information is sufficient because each node knows the paths to all nodes within its zone.

1.2.5 ARC

An Adaptive Routing using Clusters has been proposed in [BR02]. It features a cluster structure that is constructed in order to make routing more efficient.

The clusters are possibly overlapping groups of nodes, and each cluster has exactly one leader. All members of a cluster are direct neighbours of the leader, and all direct neighbours of a leader are a member of the cluster, hence the clusters may be overlapping. A node becomes a leader when it is not a member of any cluster. A leader can only lose its leader status when its cluster is a subset of another cluster, in which case the smaller cluster is abandoned in favour of the larger one. For routing, a set of gateways is established for each pair of adjacent clusters. This implies that there may be several links between two clusters.

The routes are discovered by a so-called limited broadcast. This means that the route discovery messages are not broadcast in the complete network, but only cluster leaders and gateways broadcast these messages. The use of AODV on the cluster members is proposed for the route discovery phase. The established routes between nodes are then recorded as a sequence of cluster leaders. This sequence contains sufficient information for routing the packets because each cluster leader has the knowledge of the current gateways to its adjacent cluster leaders.

1.3 ReBaC in Its First Version

The basic concept of Repair Based Clustering has already been introduced in [Hei07]. It presents a clustering algorithm for mobile ad hoc networks that is entirely based on cluster repair rules. This means that the nodes start with a simple starting state, and from there on, the cluster formation is built by the same rules that are later used to repair the clustering structure in the face of node mobility and other topology changes. The clusters constructed by the algorithm are non-overlapping.

The construction of the clustering structure is based on the cluster join functionality. When a node decides that its current situation is not acceptable

anymore, i. e. because it has lost connection to its cluster or it is a cluster head without any members, it tries to join one of the clusters that it has connection to. This information is concurrently gathered while the node is in action, and in the event of an intended cluster join, the node chooses one of the known neighbouring nodes to explicitly request membership in its cluster. The membership request may be accepted or rejected, in which case the node will try another of the known neighbouring nodes to request membership in its cluster. If no membership can be acquired, the node declares itself a cluster head in the same way as after being switched on.

The only metric that is used to limit cluster size is the number of hops towards the cluster head. A maximum number of hops is defined, and cluster membership requests are only accepted if the resulting maximum number of hops within the tree does not exceed the maximum value. On the other hand, the existence of a small cluster only terminates when there are no members in it, i. e. the cluster head is the only node in the cluster. Thus, in the worst case, there may be a large number of clusters consisting of exactly two nodes, namely the cluster head and one member.

The general approach of this scheme eliminates some of the difficulties that other clustering and routing schemes for wireless ad hoc networks have. First, if there is a different set of rules for constructing the clusters and for maintaining them later, it is more difficult to ensure that the clusters always have the properties that are intended by the clustering algorithm. Secondly, with many existing algorithms, it is assumed that there are no topology changes during the initial construction of the clustering structure. Both problems are solved with the repair based approach of ReBaC. The same set of rules is used for initial construction and later maintenance, and topology changes during construction are met with the same reactions as during the maintenance phase.

In [Hei07], simulations have been conducted showing that ReBaC is functional, but that it should be adapted to the needs of specific networks by means of changing the metrics it uses for cluster formation.

1.4 Changes in ReBaC2

The new version of the repair based clustering algorithm contains some changes that have been proposed after the evaluation of the first version of ReBaC. For example, exchangeable metrics are a key requirement to the second version. For different application and environment scenarios, different metrics are suitable to control the clustering structure. Thus, the new

version should provide the possibility to freely specify the clustering metrics independently of the algorithm behaviour. Coming along with this change, the locality of the evaluation of a cluster needs to be changed. Judging the quality of the cluster at a node that is not currently a member of the cluster, as it is in the first version, does not allow enough information to be taken into account when making decisions on further actions. This is especially true as the metrics to be used for this judgement are freely definable.

Another change since ReBaC is that a regular clustering message has been introduced in order to make the amount of messages sent by the clustering algorithm more manageable. Furthermore, the extended use of the clustering algorithm in a hybrid routing algorithm has been prepared. The first version of ReBaC was only a clustering scheme for the sake of demonstrating its feasibility, the second version enhances the demonstration to actually approach routing in mobile ad hoc networks.

1.4.1 Metrics

In the new version of ReBaC, the cluster formation is based on metrics that can be defined separately from the algorithm specification. This means that the criteria that are used to form the clustering structure are exchangeable and can be adapted depending on the needs of the application and the network situation.

The values of the defined metrics are distributedly computed by the algorithm. Decisions concerning the clustering structure are then based on these values. This is achieved by extracting the decision behaviour into a separate operator that is then defined externally with the metrics. Thus, certain conditions to the clustering structure can be specified in terms of the metrics, and these conditions can be enforced by the algorithm.

1.4.2 Cluster Join Criteria

The way the criteria are evaluated has also been changed with regards to the location of the decision. In the former version, the joining node evaluates the criterion. This criterion only consists of the number of hops to the anticipated head. In the new version, the joining node sends a jcq message as a local broadcast, thus applying for cluster membership to all neighbouring nodes. The nodes receiving the jcq message use the defined metrics to determine a value of preference for the proposed join operation from their own status. Depending on this value of preference, they may send a jca message to the

applying node. The applying node chooses one of the neighbours who sent a jca message.

The change in design most importantly means that the locality of the decision over a join operation has been moved from the joining node to the cluster member that is being queried. This allows for more information to be taken into account when deciding whether a join operation should take place. Now, all information that is present in the cluster can be used, including the externally defined metrics applied to the clustering structure.

1.4.3 Alive Messages

The meaning of the regularly sent alive messages has been slightly changed for the second version of ReBaC. In ReBaC2, an alive message is a container message that may contain other clustering messages. This concept has been introduced in order to facilitate communication with several different messages between ReBaC2 on different nodes, while at the same time only sending periodic messages over the medium.

When a node needs to send a clustering message, the message is queued until the next alive message is sent, and is then included in the alive message. This makes it possible to use various different clustering message types, while the queueing of these messages ensures that the broadcast medium will not be used too often because the different messages are transmitted together.

1.4.4 Routing

Having established a clustering structure, the next step is to devise a routing scheme benefitting from the hierarchical structure provided by clustering for efficient and, most importantly, scalable routing. To this end, the new clustering algorithm has been integrated into the routing framework presented in [FGG07]. The framework allows for a composition of different routing mechanisms and is used to combine ReBaC2 with the reactive routing scheme AODVlight to form a new cluster-based hierarchical routing scheme.

Chapter 2

Repair Based Clustering Algorithm 2 (ReBaC2)

2.1 Introduction

ReBaC2 is a distributed repair based clustering algorithm for mobile ad hoc networks. It consists only of rules repairing an existing clustering structure, and it is started by a simple rule making every node its own cluster head.

The main feature of ReBaC as compared to other clustering schemes is that it is completely repair based, i. e. the cluster formation process consists of cluster maintenance activities, even for the initial construction phase. This has the advantage that only one clustering algorithm needs to be provided, and that it produces similar results for initial construction and repair after topology change.

The additional main feature of the second version, ReBaC2, is the fully exchangeable metrics. The criteria that are used to assess the quality of the clustering structure and of join operations can be defined in a separate package.

The clusters form a (non-overlapping) partitioning of the network. Each cluster is organised in a tree structure, where the cluster head is the root of the tree, and cluster members are tree nodes or leaves in the tree. In the following, the terms *parent* and *child* will be used to indicate the relation of the network nodes with respect to this tree strucure.

Each node regularly sends alive messages. These messages are not primarily used for keeping alive existing cluster structures, but also for periodic exchange of current information. The alive messages are container messages for

all other clustering messages. They contain the node's current cluster ID and additional messages with information based on the current state of the node and the cluster. For example, a node that is currently a member of a cluster sends up and down messages in its alive message. The up messages contain the information that was aggregated in a member's subtree, and the down messages carry the information that is propagated from the cluster head to all members.

When a node changes its cluster membership, a handshake of jcq (join cluster request), jca (join cluster accept), and jcc (join cluster confirm) messages is included in the alive messages of the affected nodes. When no membership changes are occurring, every node sends one alive message with up and down messages per defined time interval, thus making the total number of messages sent linearly dependent on time. Only when cluster membership is changed do nodes send additional alive messages to request and confirm cluster joining activities.

2.1.1 Message Scenarios of ReBaC2

The following subsections show the most important communication situations using message sequence charts (MSCs). To improve the readability of the MSCs, the names of the relevant messages included in the alive messages are given rather than the alive messages themselves. The messages that are sent within a coregion are included in a common alive message that is locally broadcast over the wireless medium. Thus, these messages are actually sent in a single send operation, but received at several nodes.

For better understanding, the network topology used in each of the examples is depicted in additional diagrams. In these topology diagrams, cluster heads are coloured blue and cluster members are coloured light green or light orange. The groups of nodes encircled by a red line belong to the same cluster.

2.1.1.1 Information Dissemination

The nodes belonging to the same cluster regularly exchange information in up and down messages. The up messages are used to transmit information from the member nodes to their parent nodes and eventually to the cluster head, and the down messages are used to disseminate central information from the cluster head to all members of the cluster.

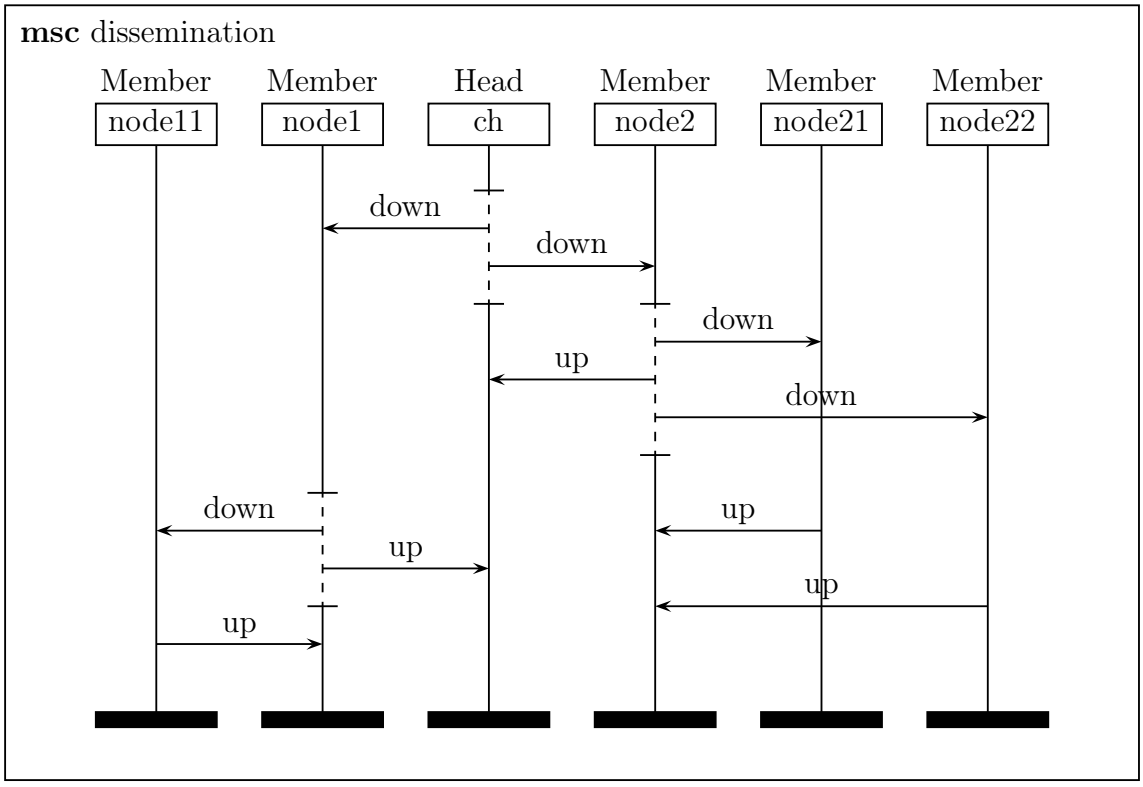

Figure 2.1: Information dissemination

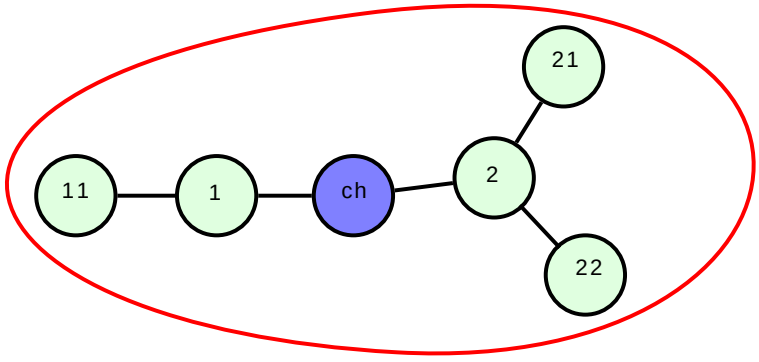

Figure 2.2: Network topology for information dissemination MSC

The message sequence chart in Fig. 2.1 shows some nodes of the same cluster and their regular exchange of information. The network topology for this example is depicted in Fig. 2.2.

The sending of an alive message is triggered at a node as soon as a down message can be sent. The cluster head sends alive messages with down messages regularly. A node that is a member of a cluster immediately forwards a down message it receives. Thus, the regular alive messages in the complete cluster are triggered by the cluster head. This effect is used to make sure that the information from the cluster head is spread across the whole cluster within a short time. It is not intended to be a time synchronisation of the nodes in the cluster.

When the sending of an alive message is triggered, all other pending messages are included in the alive message and sent. By retaining the non-urgent messages and incorporating them into the next triggered alive message, a more efficient use of the broadcast medium is attempted, as there are fewer messages in total that need to be sent over the wireless medium. This is true even if the resulting message has to be fragmented into several frames on the medium, because the frames of the transmission can be more optimally utilised if one large message is sent. As the alive messages are received by all direct neighbours, each neighbour can use the messages intended for it and discard the other messages included in the alive message. The up messages are typically such pending messages that are sent together with the down messages when the sending of a message is triggered.

2.1.1.2 Cluster Join

The MSC in Fig. 2.3 shows the message sequence in case a node joins a cluster. The joining node is called `node0` in the diagram. The network topology is shown in Fig. 2.4.

The first action of a node that has decided to join another cluster is to broadcast a jcq (join cluster request) message. This message is processed by all direct neighbours of the node. Each of the recipients of the jcq message computes a value of preference for the join operation and, if the value is good enough, offers cluster membership to the new node by sending a jca (join cluster accept) message. The new node uses a timer to wait for incoming jca messages for `jcaTime`, and after that time, it chooses the jca message with the best preference value, and sends a jcc (join cluster confirm) message in reply to its sender. This message also includes information on the subtree that the joining node may already have. If this is the case, the node joins the new cluster with its subtree. The accepting node (`node12` in the diagram)

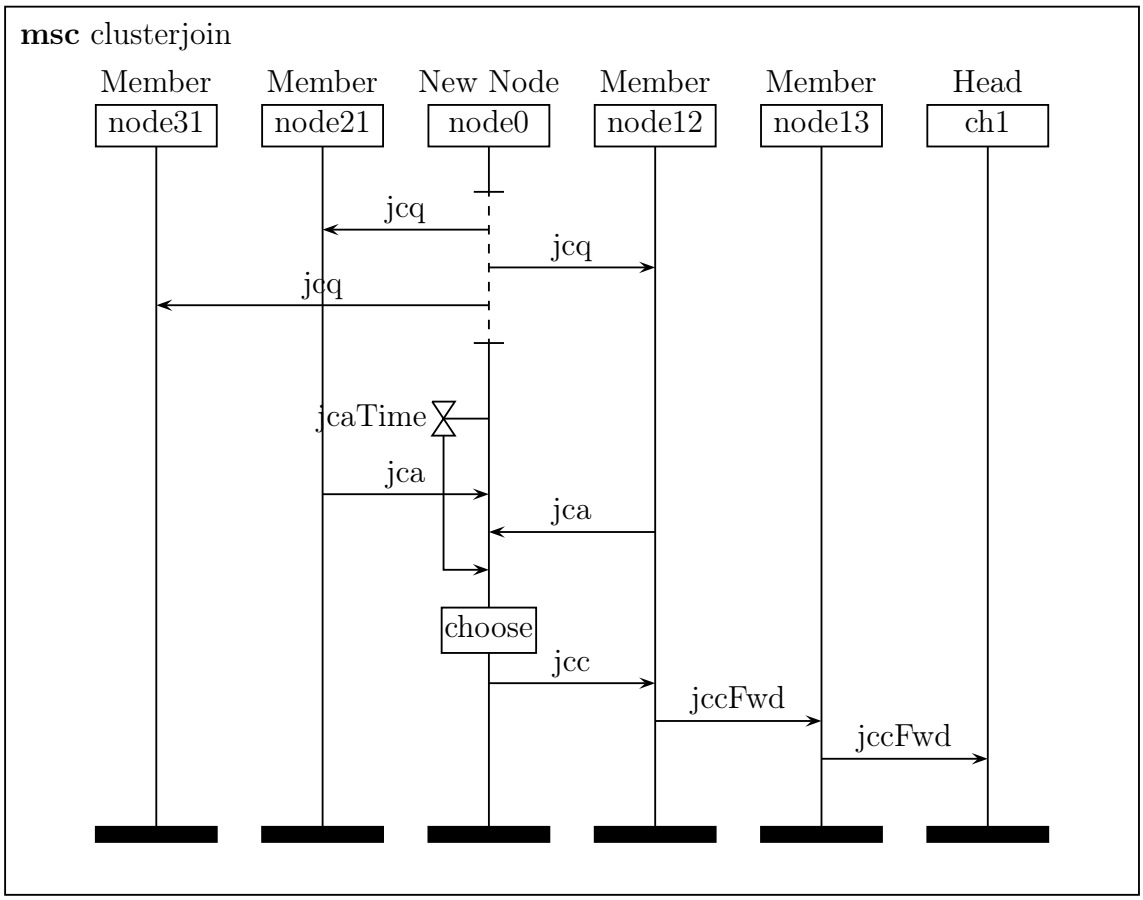

Figure 2.3: Cluster join

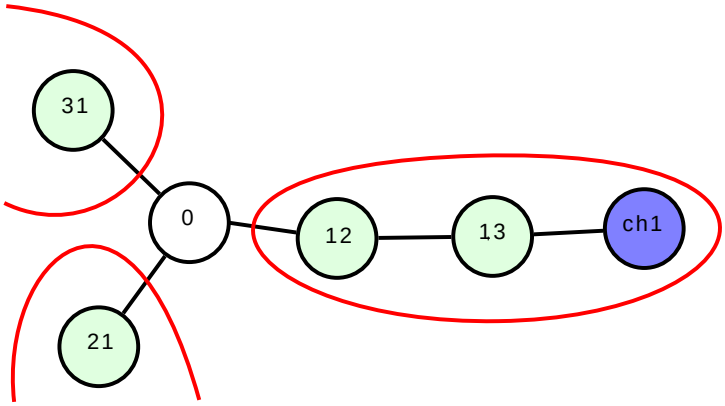

Figure 2.4: Network topology for cluster join MSC

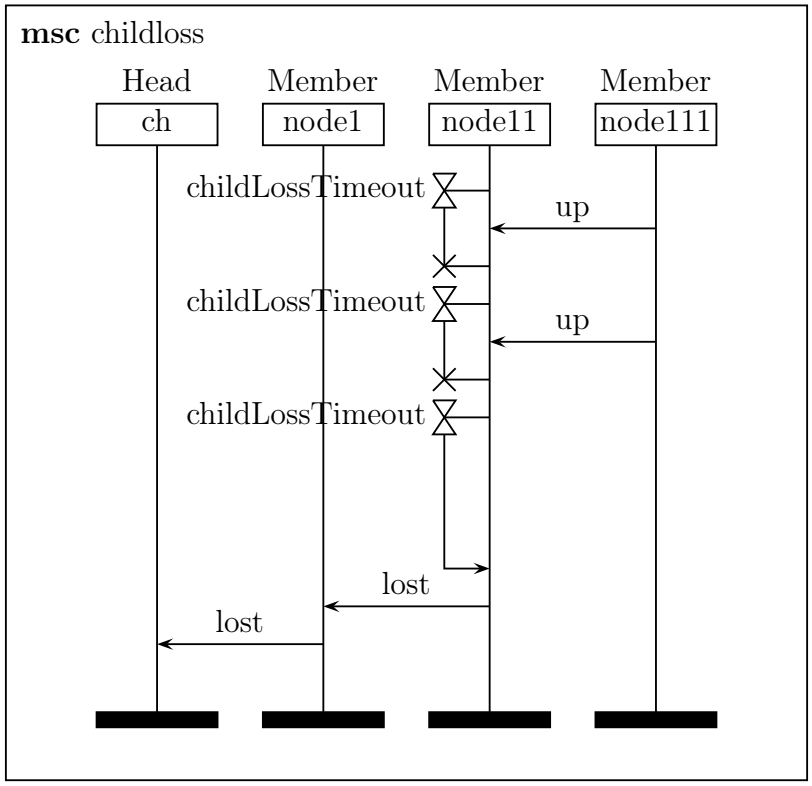

Figure 2.5: Child loss

forwards the information of the newly joined node towards the cluster head in a jccFwd (forwarded jcc) message. For forwarding the information towards the cluster head, jccFwd messages are used instead of jcc messages because they contain additional fields for the current sender and receiver of the link the message is forwarded over.

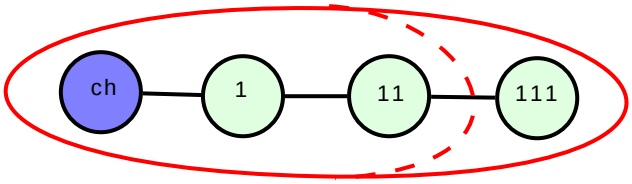

Figure 2.6: Network topology for child loss MSC

2.1.1.3 Child Loss

The MSC in Fig. 2.5 shows how the loss of a child is detected and the reaction to it. The topology can be seen in Fig. 2.6. The continuous red line encircles the cluster as it is before the child loss occurs, and the dashed line shows the change in cluster membership after child loss.

In normal situations, when topology is stable, the periodic up messages included in the alive messages are used to confirm the existence of a child. Thus, as soon as an up message is received from a child, a child loss timer is restarted for that child. The timer duration is set high enough so that up to two missing up messages are tolerated without a formal child loss detection. When the timer times out, it is assumed that the child has been lost, and that it is not a member of the cluster anymore. In case the lost child has children, they are also considered lost, because they are not reachable anymore if their parent node has gone. After the timeout, a lost message is generated and sent towards the parent node of the node that detected the loss. The message is forwarded towards the cluster head. All nodes on the way purge the lost node and all its direct and indirect children from their internal representations of the subtree structure. When this process is completed, the lost node has been completely removed from all data structures of the cluster. The lost child node also discovers the link loss with a timeout and tries to join a cluster in its vicinity.

2.1.2 Interaction in Special Cases

When a node's direct or indirect parent decides to change its cluster membership (and that of its children), its children have to be notified of the change in cluster membership. The parent sends its new cluster ID with its down messages, and its children adopt the new cluster ID when receiving a down message from their direct parent. In the same way, the children's children are notified of the changed cluster ID.

When a child seems lost for a while, but later reappears, its subtree information has already been purged from all intermediate nodes (see MSC in Fig. 2.5). If the child itself still considers itself a member of the cluster, for example because it is still within reach of its parent, it will be reaccepted into its previous cluster as soon as a message from the child is received at the parent. To regain the already purged subtree information from the rediscovered child, the parent resends a jca message towards the child as soon as it rediscovers a previously lost child node. The child node resends a jcc message with its current subtree structure on receipt of a duplicated jca message, in

the same way as if it had just joined the cluster as shown in Fig. 2.3. The jcc message is forwarded towards the cluster head by the parent, just as a jcc message from an initial join action is forwarded, so that all affected nodes gain knowledge of the child's subtree structure.

2.2 SDL Design of ReBaC2

ReBaC2 has been specified in the *Specification and Description Language* SDL-92, which will be called *SDL* in the following sections. The specification has been done with the tool *Telelogic Tau* [tau].

2.2.1 SDL Block Structure

The SDL block type `ReBaC2` (see Fig. 2.7) contains the clustering functionality provided by ReBaC2. On its borders, it uses the interfaces provided by the framework discussed in Section 3.2. The interface `RInf` is used to exchange information with other nodes on the network, and the interface `MGMNT` is later used for querying ReBaC2 about routing information.

The SDL block `ReBaC2Adapter` contains coder and decoder functionality that is extended by the ability to incorporate information obtained by measurements taken when a frame is received on the wireless MAC layer, e.g. transmission quality, into the signal localAlive. It decodes incoming RecvData signals into alive messages that contain the specific clustering messages. The alive messages are delivered to `AliveReceiver`. The coder part of the functionality lies in encoding the alive messages coming in from `AliveSender` into SendData signals that are then delivered to the underlying transport mechanism for being sent via the wireless medium. The additional measurements that are included into the decoded localAlive signals are obtained from the wireless interface when the corresponding data frame is received. Currently, these measurements consist only of a value for the transmission strength of the frame at the receiver side. It is used to assess the quality of the wireless link. This value is included with the decoded alive message into the localAlive signal that is sent to `AliveReceiver`.

The block `AliveSender` sends alive messages regularly. It collects all messages that are received from block `Control` and includes them in the next alive message. The block `AliveReceiver` decomposes the incoming alive messages and sends the messages contained in them to `block Control` individually.

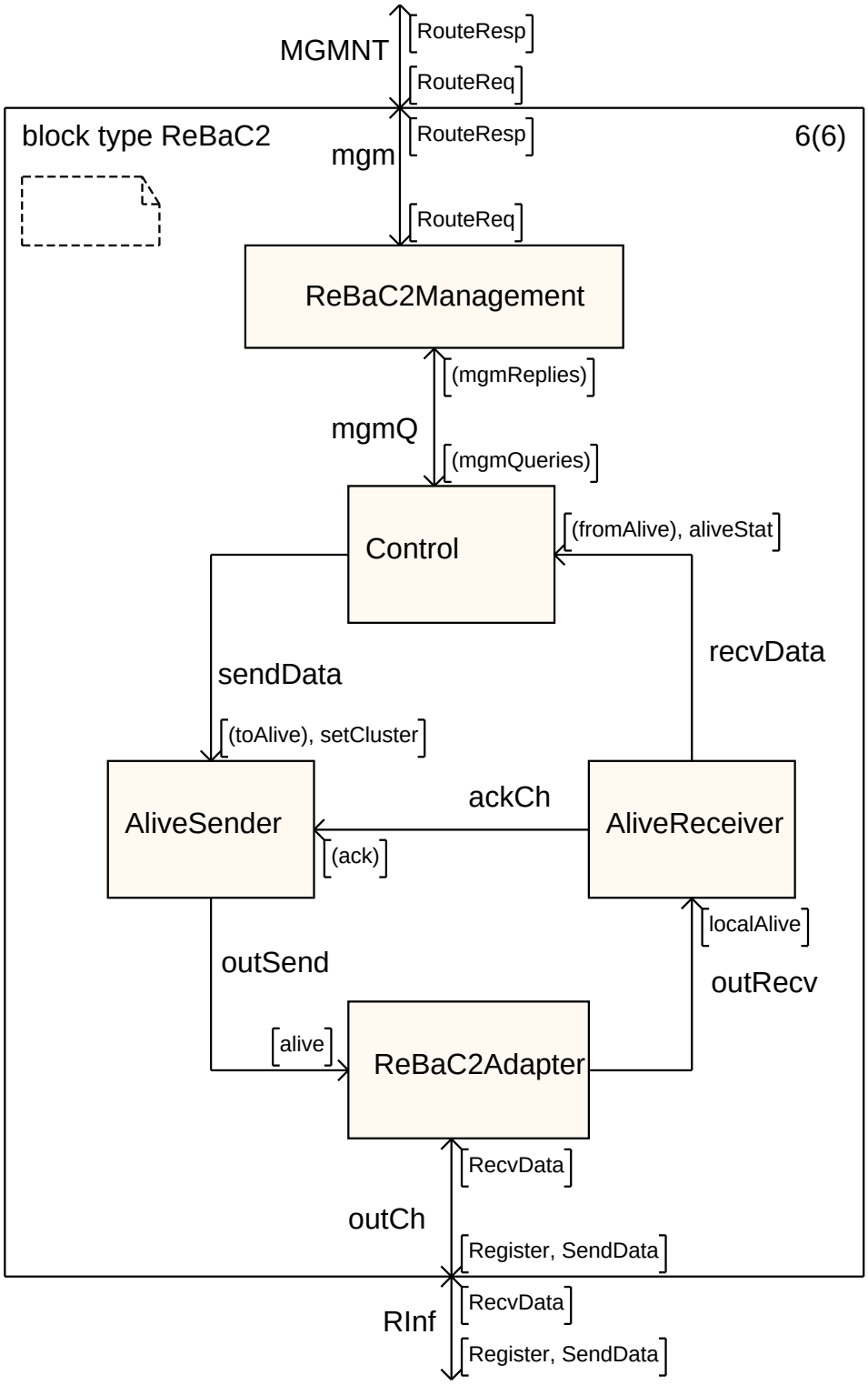

Figure 2.7: Block type `ReBaC2`

The block `ReBaC2Management` is responsible for answering route requests from outside of ReBaC2. Each incoming route request is broken down into requests for single nodes which are sent to service type `Route` in block `Control`. The resulting route response is formed from the answers from block `Control`.

The block `Control` consists of exactly one SDL process, `ControlServices`, which in turn consists of the five SDL services explained in Section 2.2.2. In the interaction of these services, and in their communication with their environment outside the process lies the clustering functionality.

2.2.2 Behaviour

The main process, `ControlServices`, consists of the following services:

- `Cluster`. Cluster membership establishment and management
- `Evaluate`. Evaluation of incoming cluster join requests
- `Aggregate`. Aggregation of subtree information for collected relay to parent node
- `Observe`. Observation of local properties and of the node's neighbourhood, including processing of the parent's down messages
- `Route`. Response to routing requests using locally available cluster structure information

The following sections detail the interaction of the processes and services within a node for some of the more common communication scenarios.

2.2.2.1 A Cluster Is Joined

The message sequence chart in Fig. 2.8 shows the local communication within a node when another node joins the cluster via this node. In the beginning, a localAlive message containing a jcq message is received. The *local* variants of some messages have been introduced for local communication within the node because they contain additional reception information for these messages, such as for example transmission strength. Thus, the jcq message is delivered internally as a localJcq signal. Upon receiving a localJcq signal, service `Evaluate` computes a preference value for the proposed join operation. As the jcq message also contains aggregated information on the joining node's subtree, this information can be taken into account when computing

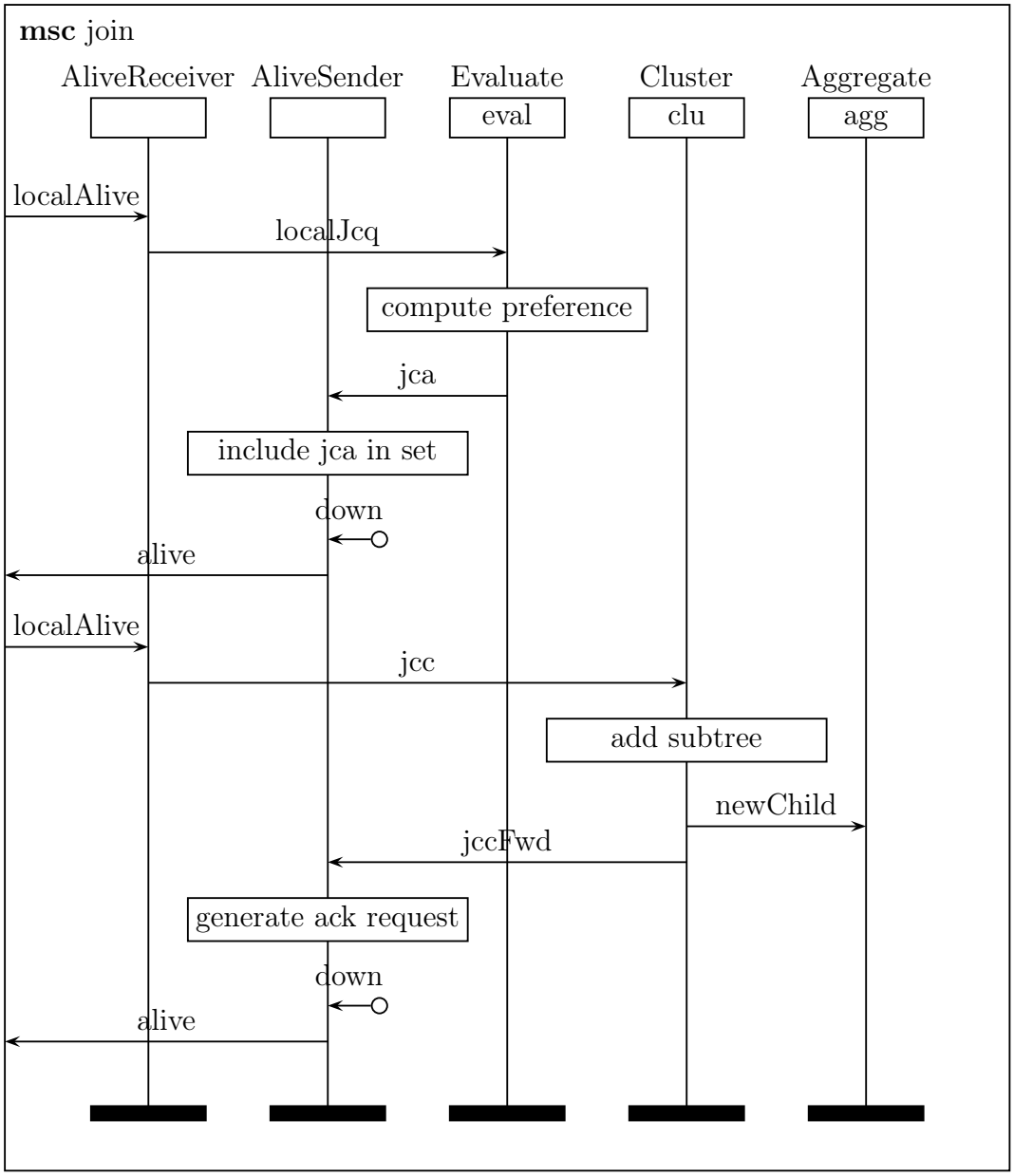

Figure 2.8: Evaluation of incoming jcq at a member node and another node joining this cluster

the preference value. For example, if a subtree with two hops depth is trying to join, but the maximum number of hops allows only one hop to be added, the join operation may be rejected. To reject a join operation, the node simply does not answer the jcq message. Thus, service `Evaluate` only replies to the localJcq signal if the value of preference is good enough. The answer consists of a jca message that is collected by process `AliveSender`. It is not directly sent, but only as soon as another outgoing message triggers the sending of an alive message. Thus, in the MSC, a down message of irrelevant origin is included to show that the sending of the alive message to the medium is triggered.

After receiving the alive message, the other node decides which cluster to join. In this case, the other node chooses this node as its future parent, so a localAlive signal containing a jcc message is received. The jcc message contains the subtree structure information from the joining node, so this information can now be added to the locally available subtree structure information by service `Cluster`. Additionally, service `Cluster` sends a newChild signal to service `Aggregate`. This is necessary to inform service `Aggregate` of the new child this node has, because it is part of the responsibility of service `Aggregate` to keep track of all direct children of this node and to detect the loss of a direct child. After sending the newChild signal, service `Cluster` sends a jccFwd message that is intended for the parent node of this node. This message includes the new child's subtree structure information so that all parents of this node, and eventually the cluster head, can add this information to their local representations of their subtree.

As it is important for the jccFwd message to be received by the parent node, this message type has been designed to be a dependable message. This means that additionally to the message itself that is added to the set of messages to be sent with the next alive message, a seqNo message is included in the set. The seqNo message has the function of an acknowledgement request. It contains a sequence number unique to this node, and when it is received by the intended recipient node, an ack message is sent back in reply. This functionality is explained in more detail in Section 2.2.2.4.

Again, the alive message is not sent until some other message triggers it, and then the information is sent out to the node's parent.

2.2.2.2 Cluster Merge

Fig. 2.9 shows an example situation of two clusters merging. The solid red lines show the cluster structure before the merge operation. Node `n221` has been lost by cluster 2 (the small cluster to the right). The cluster head

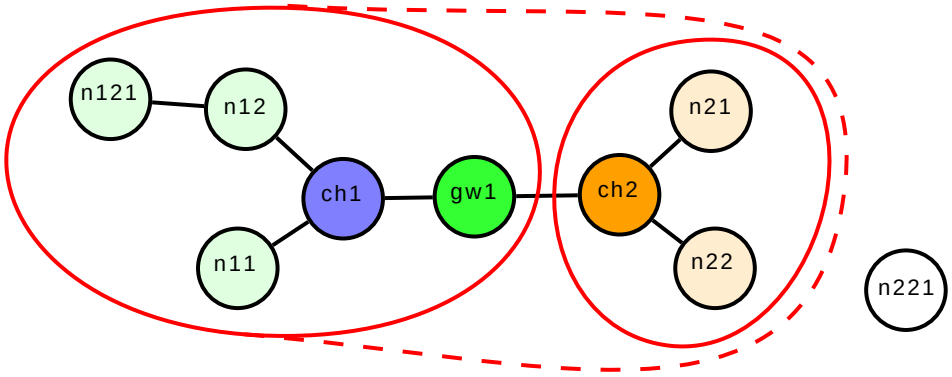

Figure 2.9: Example topology for cluster merge scenario

of cluster 2, `ch2`, recognises that its cluster is *too small*, i. e. not meeting the given criteria, and consequently joins the neighbouring cluster 1, thus merging its cluster with cluster 1 and abandoning its cluster head status. The resulting cluster structure is shown by the red dashed line.

The message sequence chart in Fig. 2.10 shows the internal signals in the cluster head in the situation when it is informed about the loss of a node that makes the cluster too small. The scenario starts with an alive message that contains a lost message, i. e. it informs the cluster head that it has lost a node (and its subtree). The alive message also includes a seqNo message requesting an acknowledgement of its reception. In reaction to the seqNo message, process `AliveReceiver` sends an ackSend signal directly to process `AliveSender` which sends the acknowledgement message with the next alive message. The lost message is delivered to service `Cluster`, which truncates the lost node's subtree from the local cluster representation. After doing so, it uses the operator `clusterTooSmall` to determine whether the existence of the reduced cluster still conforms to the criteria laid out in the metrics definitions. In this case, the cluster is determined to be too small, and the timer `noChildrenTimeout` is used to wait for a while for other nodes to join this cluster, until the cluster is finally merged with another cluster. In this case, no other nodes join the cluster during that time, and after that time, the cluster head starts joining another cluster, which means that it is taking its members with it into the new cluster. The further actions taken by the head in order to join another cluster are described in Section 2.2.2.4.

2.2.2.3 Parent Loss

In Fig. 2.11, it is shown how a member node detects the loss of connection to its parent and how it reacts. The normal situation, i. e. the situ-

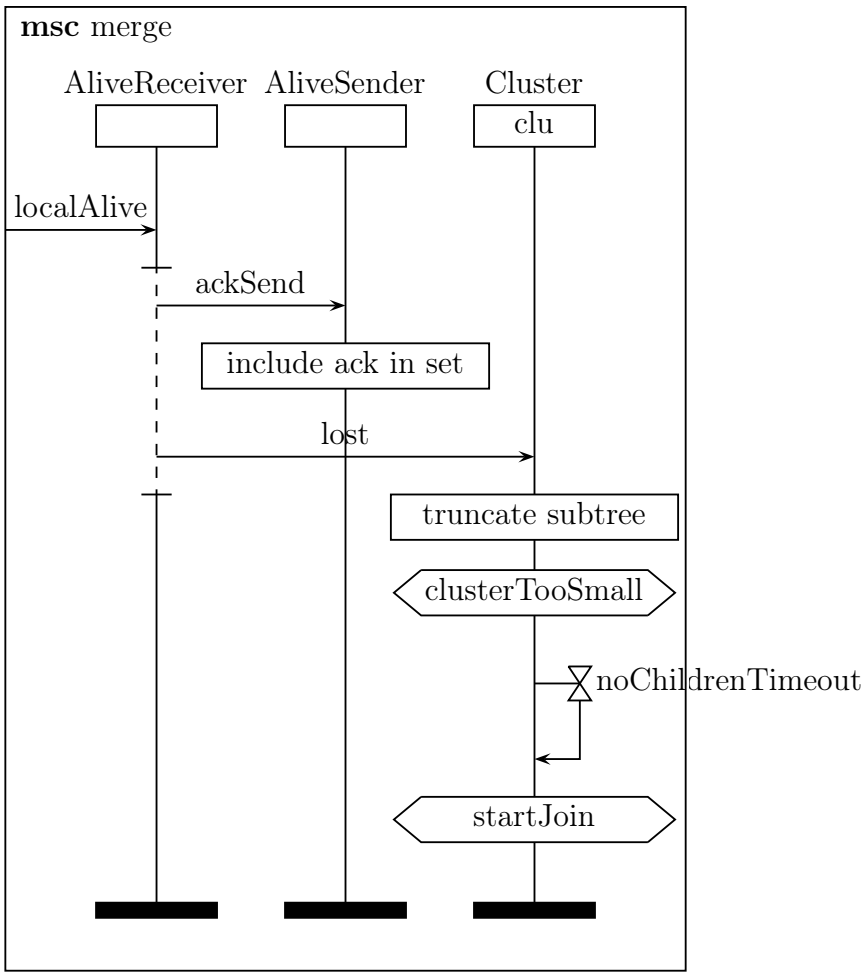

Figure 2.10: Cluster head deciding to merge its cluster with another cluster

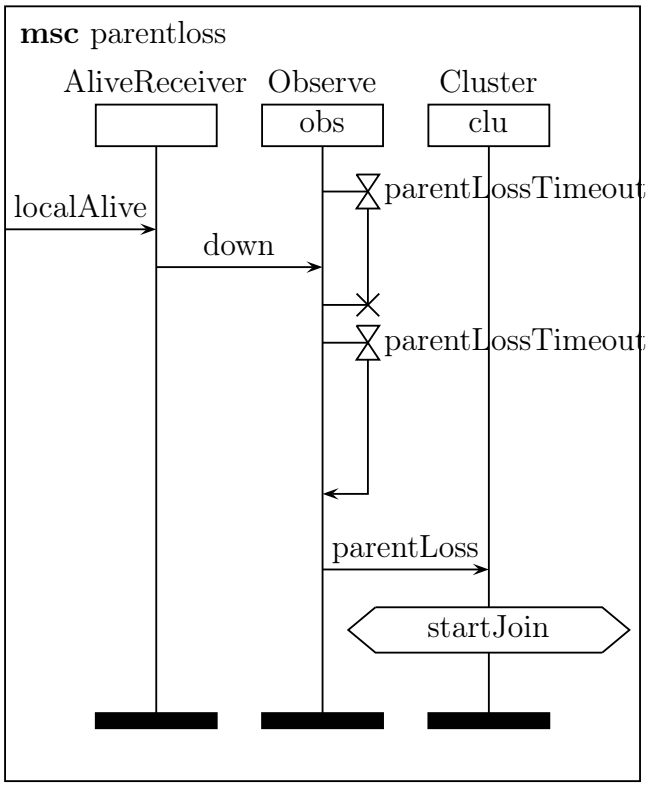

Figure 2.11: Parent loss at a member node

ation where the connection to the parent is stable, is shown in the beginning. The down message from the parent node is received before the timer parentLossTimeout times out. When the down message is received, the timer is stopped and restarted. In case the timer times out, the connection to the parent is assumed to have been lost, and service Observe sends a parentLoss signal to service Cluster to inform it of the parent loss. In service Cluster, the functionality for joining another node with the complete subtree is started. This behaviour is explained in Section 2.2.2.4.

2.2.2.4 Joining Another Cluster

The activities performed by a node that has decided to join another cluster are shown in the MSC in Fig. 2.12. The first action taken by service Cluster is to inform process AliveSender of the fact that this node currently belongs to no cluster. This is necessary because process AliveSender sends the current cluster membership information of this node with all alive messages. As the messages used to obtain a new cluster membership are also included in alive messages, it is important that these alive messages do not contain the previous cluster membership anymore. After that, service Cluster sends the jcq message that is broadcast to all neighbouring nodes, and uses the timer jcaTime to determine the time that the other nodes have to send in their replies. The jcq message immediately triggers the sending of the alive message at process AliveSender, so the message is immediately sent to the other nodes.

The incoming jca messages are collected at service Cluster until jcaTime is over. Then, the jca message with the best preference value is chosen, making its sender the future parent of this node. The future parent node is informed of this node's choice with the jcc message that is explicitly addressed to it. The jcc message contains the information on this node and on its subtree. As it is vital for the consistency of the nodes' views of the clustering structure that the jcc message is not lost on the way to the new parent, an acknowledgement request is added for this message at process AliveSender. The jcc message triggers the immediate sending of the alive message so that the information on the new members of the cluster is not deferred.

This MSC also shows how the dependable message delivery is organised. In case a dependable message is not acknowledged until the next alive message is sent, process AliveSender sends it again with the next alive message, of course also including the seqNo message requesting the acknowledgement of the message. This is repeated with every alive message until the acknowledgement from the intended recipient of the message is received. When

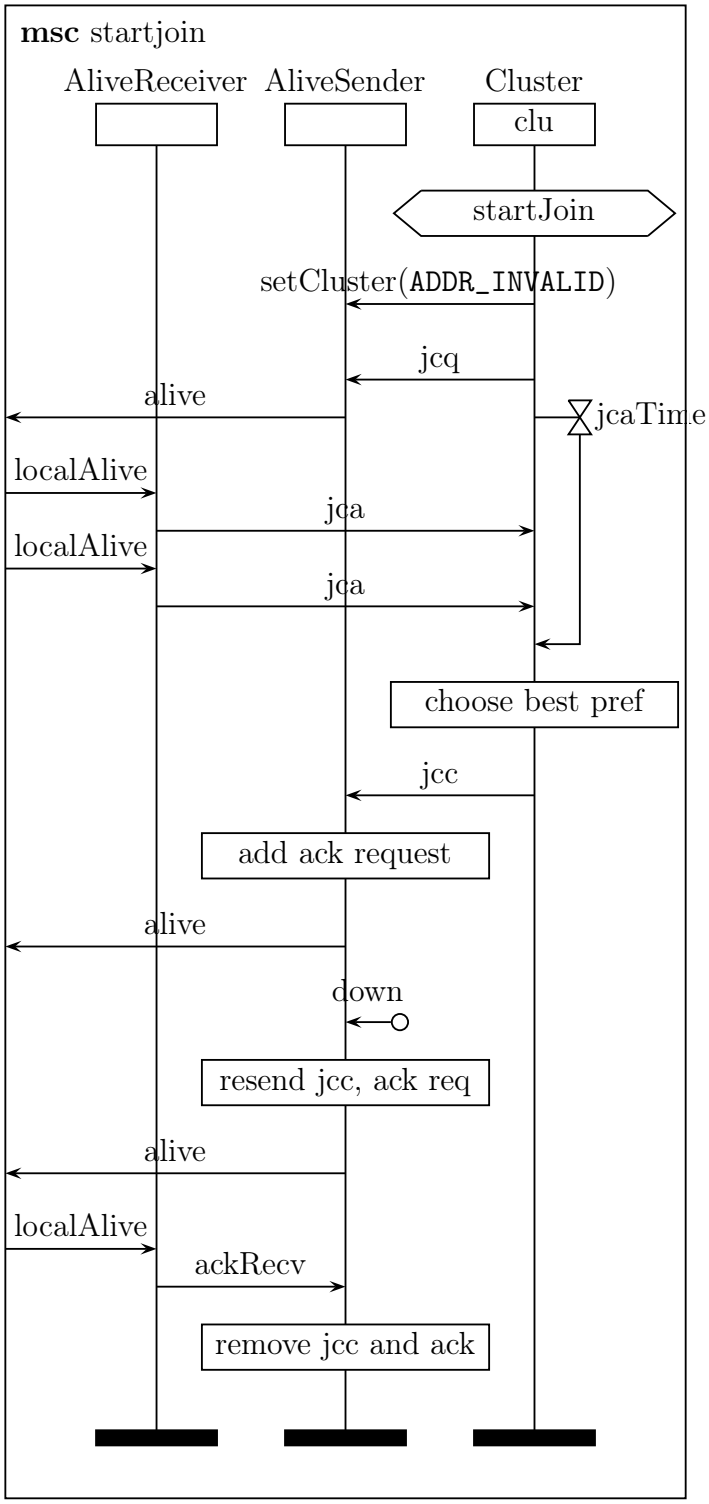

Figure 2.12: This node joining a cluster, with repeated sending of dependable jcc message

the acknowledgement is received, process `AliveReceiver` sends an ackRecv signal directly to process `AliveSender`, notifying it that the message has been acknowledged and that it does not need to be sent anymore. Process `AliveSender` thus removes the message and its seqNo message from the list of unacknowledged dependable messages.

2.2.3 Process `ControlServices`

In the following five subsections, the services of process `ControlServices` are described in detail. For an overview of their interaction channels, see Fig. 2.13. The last subsection details the mechanism for dependable delivery of some types of clustering messages.

2.2.3.1 Service Type `Evaluate`

This service evaluates incoming jcq messages that are received from a neighbouring node wishing to join a cluster. The jcq message is evaluated by all receiving nodes, and if the quality of the cluster resulting from the supposed join action is good enough, the neighbour sends a jca message including the value of preference of the cluster join action. The jcq messages are received by the service as localJcq signals because that signal carries additional information on transmission quality of the message. When a jcq message is received at the service, the function `evaluate` from the package `Metrics` is used to determine the preference value for the requesting node. If the preference value is better than a threshold also defined in package `Metrics`, a jca message is sent in reply to the requesting node.

2.2.3.2 Service Type `Observe`

The functionality of `Observe` is split up among three service definitions. The service type `Observe` inherits service type `ObserveNeighbourhood`, which inherits `service type ObserveLocally`.

`ObserveLocally` is used for the information that is observed locally within the node without regard to other nodes. Examples include an assessment of battery power and expected remaining operating time. Currently, it also includes the tracking of the number of child losses within the last period of time. This is considered local information because this service is notified locally by service type `Aggregate` about single incidents of child loss. The information that is acquired is stored in the local variable `localMetric` that

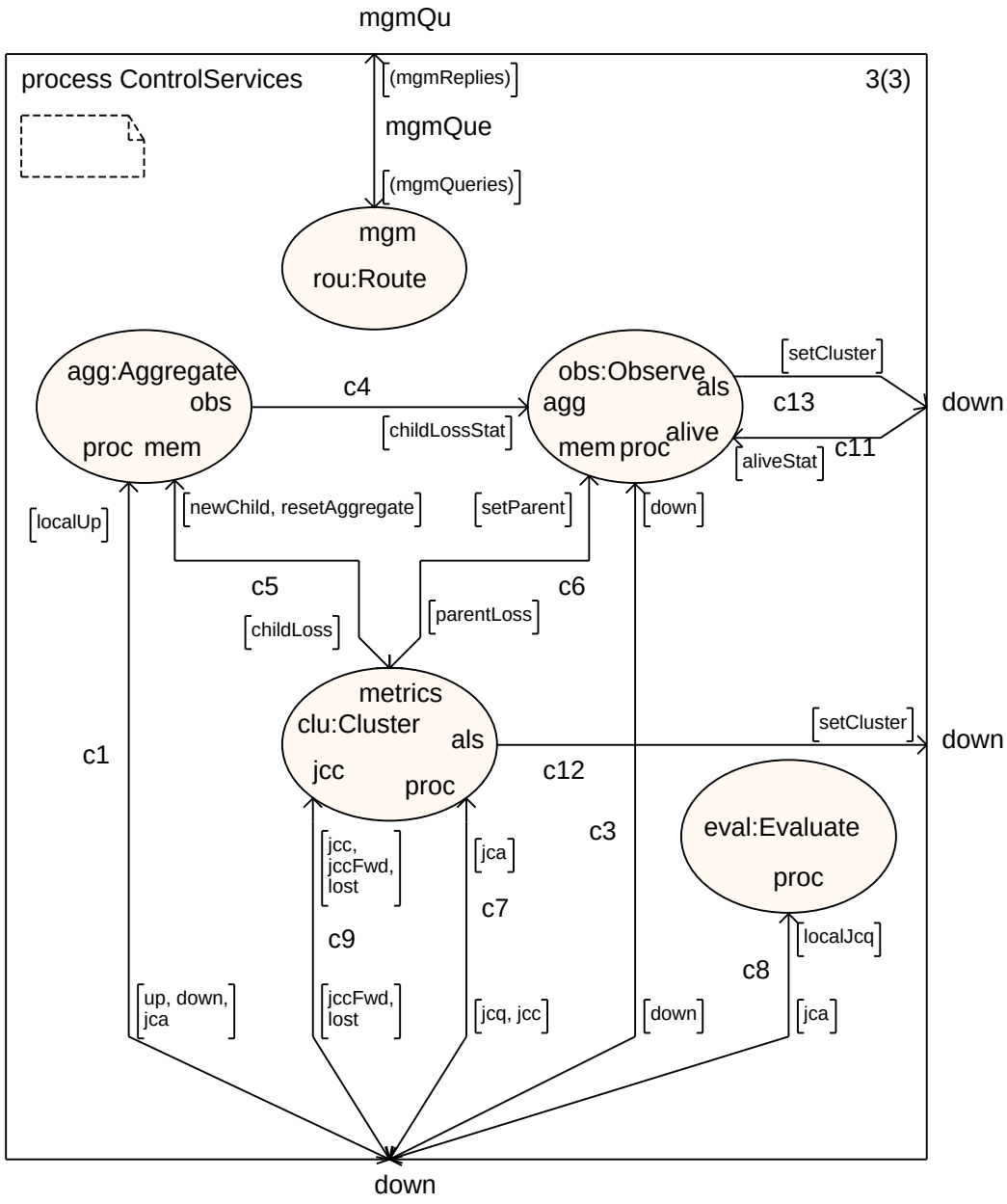

Figure 2.13: Service interaction page of process `ControlServices`

is defined in process ControlServices and is thus visible by all services of the process. This visibility is used for information exchange between the services of the process.

ObserveNeighbourhood receives an aliveStat signal from AliveReceiver for every alive message that has been received by this node. As alive messages are only received from direct neighbours of this node, the service monitors the direct neighbourhood of the node. It may be used to compute values such as the number of direct neighbours that belong to the same cluster, or the change of transmission quality of each node as compared to the previously received frames. The functionality is also used to monitor the direct neighbours of the node that are *not* members of this node's cluster. All such nodes are considered as potential gateways to their cluster, and this node is then also a gateway node for this cluster. The information on neighbouring gateway nodes is also saved in the variable localMetric. Additionally, the service contains a timeout mechanism that removes gateways from the set of gateways in localMetric when their alive messages have not been received for too long a while.

Observe inherits the two other observe services, and offers additional functions such as forwarding of down messages and recognition of parent loss. Incoming down messages are only forwarded when this node is a member of a cluster, i. e. it is not a cluster head. The down messages originate from the cluster head and contain cluster-wide information that is released by the cluster head and then propagated through the tree structure by all cluster members. Thus, every member node forwards down messages it receives from its parent to its children. At the same time, service type Observe detects parent losses, i. e. the absence of several consecutive down messages from the direct parent. When there has not been a down message from the node's parent node for a while, the parent node is considered lost, and service type Observe sends a parentLoss signal to service type Cluster which takes appropriate actions.

There is another additional function of Observe which is needed when a (direct or indirect) parent of this node changes cluster membership. When the node's parent changes its cluster membership, it includes the new cluster ID in its down messages (see Section 2.1.2). Such a change of cluster ID is detected at Observe, and a setCluster signal is sent to AliveSender, notifying it of the changed cluster membership for further propagation to this node's children. The information on the new cluster ID is also stored in the local variable clusterId for the other services in the process.

2.2.3.3 Service Type Aggregate

Aggregate aggregates incoming up messages from the children of this node. The up messages contain information on the subtree of the sending node. In service type Aggregate, these up messages are collected and, after a specified interval of time, they are aggregated into one up message to be sent towards this node's parent. The actual aggregation of the metrics is performed by using the operator aggregate specified in package Metrics. This aggregation process also takes into account the information from the local reception of the up messages being aggregated, e. g. transmission quality, which is included in addition to the actual message content in localUp signals. The service also keeps track of the current set of children. It is informed by Cluster about new (direct) children with a newChild signal and uses a separate timer for each child to determine whether the connection to the child can be considered lost. In case a child is considered lost, a childLoss signal is sent to Cluster.

In the special case where this node is a cluster head, the incoming up messages are aggregated but the aggregated information is not sent as an up message, but it is converted into a down message to be propagated among all members of the cluster. A down message contains nearly the same information as an up message, but some information may be left out. For example, the cluster head receives the full list of all gateway nodes in the cluster in its up messages, but this knowledge is not required at all member nodes, so this information is not included in the down messages.

Another special case is the following. When an up message is received from a node that is not currently known to be a child of this node, but it considers itself a child of this node, (which can be seen from its cluster ID and its parent ID) then Aggregate sends an additional jca message to this specific child. On receipt of this additional jca message, the child node resends a jcc message with the complete subtree information, effectively rejoining the cluster via this node. The incoming jcc message is processed by Cluster.

2.2.3.4 Service Type Cluster

Cluster is the service with the central cluster membership and cluster head functionality. The service declares the node a cluster head when started.

As a cluster head, a node receives the jccFwd and lost messages from its members and builds an internal representation of the cluster's tree structure. The cluster head monitors the current quality of the cluster, and if the quality continues to be too poor for a certain amount of time, it takes actions to improve the quality of the clustering structure. As an example, when the

total number of nodes in the cluster is too low, the cluster head tries to join another cluster, taking the old cluster members into the new cluster as its subtree. Another example of the node starting to join another cluster is when this node is a cluster member, but has lost the link towards the parent node and thus towards the cluster.

The action of joining another cluster is started by locally broadcasting a jcq message. If there are neighbours willing to take this node as a child, they will send a jca message with a preference value to this node. The incoming jca messages are collected by `Cluster` until `jcaTime` is over, and the sender of the jca message with the best preference value is chosen as the new parent. A jcc message is sent to the new parent which includes the current subtree structure, so all members are taken into the new cluster with this node. This is a relatively large message, but it is only sent once when this node joins another cluster. Consecutively, only current metrics information is propagated towards the parent node in up messages. If no jca messages have been received after the sending of the jcq message, the node has to stay or become a cluster head for its subtree structure. Note that the node's subtree structure may be empty, resulting in the node being alone in its cluster.

When this node receives a jcc message from a node that has earlier on received a jca message from this node's service type `Evaluate`, i. e. has been accepted for joining, it merges the membership structure information from the incoming message with the locally available structure information, thus updating the local structure information to reflect the complete subtree of this node. Additionally, this node's service type `Aggregate` is informed of the new child with a newChild signal, and the contents of the jcc message are forwarded as a jccFwd message towards the cluster head. All nodes on the way to the cluster head and the cluster head itself use the presented information to update their subtree structure information. Thus, all nodes between the newly joined node and the cluster head keep a correct view of the subtree structure. In particular, the cluster head as root of the tree knows all cluster members and paths. Similar to the propagation of jccFwd messages, when a child is considered lost, `Cluster` receives a childLoss signal from `Aggregate`. In this case, it purges the lost node and its complete subtree from the subtree structure, and sends a lost message towards the cluster head. All nodes on the way to the cluster head and the cluster head itself do the same, so that the knowledge of the lost node is completely removed from the cluster. The handling of jccFwd and lost messages, i. e. updating and forwarding of local information, is also performed in `Cluster`.

When a parentLoss signal is received from Observe, or the membership structure of the cluster is unsatisfactory, e. g. because there are too few members, after a while, this node tries to join a cluster by sending a jcq message. If there are some jca replies, the node joins the cluster that sent the best reply with its complete subtree, sending the subtree information to its new parent with the jcc message, and propagating the changed cluster ID to its children with the regular down messages. If there are no jca replies at all, the node declares itself a cluster head with the nodes in its subtree as its members.

2.2.3.5 Service Type Route

Route uses the information stored in the local variables of process ControlServices to answer routing queries from ReBaC2Management. This allows for a usage of the clustering scheme as a locally confined proactive routing scheme. Specifically, information on the subtree structure, the path towards the cluster head, and gateway nodes is used by Route. This information is provided by the other services of the process. The queries between ReBaC2Management and Route are simple queries that are structurally adapted to the clustering scheme. For instance, the next hop towards the cluster head (signal getNextHopToHead), or the local and remote gateway nodes towards a given cluster ID (signal getGw) can be requested. The signal getNextHopsToTarget requests a sequence of node IDs that make up the path to the specified target, but this sequence is only returned if it is obtainable from the locally available information at this node, such as the subtree structure or gateway status of known nodes. If the node is a cluster head, the list of neighbouring cluster heads can be requested with a getAllChs signal. A cluster head is considered to be neighbouring when a gateway to its cluster is known. The list of neighbouring cluster heads can be used to request a gateway to each of them individually.

To sum this up, a member node can provide the following routing information:

- Link to direct neighbours such as parent node, children, or remote gateways when this node is a local gateway
- Complete route to all member nodes in the node's subtree
- Next hop towards local cluster head

As a cluster head, a node is additionally able to provide the following information:

- Complete route to all active remote gateways of the cluster
- List of neighbouring cluster heads and partial routes to them; the routes are provided up to the remote gateway nodes, from where the route towards the local cluster head is known.

2.2.3.6 Dependable Message Delivery

There is an acknowledgement scheme that is specified between `AliveSender` and `AliveReceiver`. Some of the messages that are sent by block `Control` require dependable delivery to the intended recipient node, i. e. message loss is not acceptable for these message types. It is assumed that an alive message is either received completely or not at all, as the underlying layer only delivers error-free messages. Thus, for each dependable message that is sent in an alive message, a unique seqNo message is also included in the alive message. On receipt of a seqNo message, block `AliveReceiver` on the receiving side generates an ack message to be sent back to the sender of the original message with the next alive message. In block `AliveSender`, all unacknowledged dependable messages are resent in every regular alive message until their acknowledgement is received.

The messages that require dependable delivery are:

- jcc - Join Cluster Confirm
- jccFwd - Join Cluster Confirm Forwarded
- lost - Node Loss Notification

These are the three message types that propagate information on new or lost members through the cluster towards the cluster head. When a node joins the cluster, its jcc message to its new parent will start the immediate propagation of this information in jccFwd messages through all nodes on the way to the cluster head. The arrival of these messages must be guaranteed in order for the cluster head to be able to have a correct view of its cluster members and structure. Similarly, lost messages are generated at the parent node who has lost a child, and need to be reliably forwarded towards the cluster head so that the information on cluster members and structure remains consistent and as accurate as possible at all times.

2.3 Metrics

The metrics used for cluster formation and evaluation are defined in package `Metrics`. This includes the definition of data structures representing the different measurements made in the cluster, as well as the operators evaluating these measurements. These definitions are separated from the main algorithm specification in order to make them more easily exchangeable.

2.3.1 Data Structures

There are several different data structures that are used for storing metrics information. These data structures are presented in the following paragraphs.

`DownMetric` includes the information that is sent from the cluster head to all its members. It currently includes the total number of nodes in the cluster, the maximum hop distance between a member of the cluster and the cluster head, and a consolidated assessment of the communication quality in the cluster. This metric also includes a field with the number of hops that a member node is away from its cluster head. This information is provided by incrementing the value each time a down message is forwarded by a member hop.

`UpMetric` contains the information that is sent from every member node towards its cluster head. It contains aggregated information about respective subtrees in the cluster. This may for example include the depth in hops of the subtree, the number of nodes in the subtree, or the set of gateway nodes in the subtree.

`LocalChildMetric` contains the additional information that is gained when receiving a frame. Currently, it consists of a value for the communication quality which is computed from the signal strength of the incoming frame. This value is used together with `UpMetric` to assess the expected quality of the cluster after a join operation that includes the given link into the cluster tree structure.

`ChildUpMetric` combines the two types `UpMetric` and `LocalChildMetric`. This data structure is needed by service type `Aggregate` when several received messages with reception information are collected for later aggregation.

`LocalMetric` contains information that is locally available at the node, e. g. the node's battery status or the direct neighbours of the node that belong to other clusters.

2.3.2 Operators

The operator `aggregate` is used for aggregating the information of the node's subtree. It takes into account a set of `ChildUpMetric` representing the information from all children in the cluster tree and it also uses the locally available information from `LocalMetric`. The operator generates a value of `UpMetric` representing the node and its complete subtree. That value is then sent to the node's parent node. As an example, the operator computes the maximum number of hops of all subtrees and adds one, using this value as its own maximum number of hops in the subtree, or it adds up the numbers of nodes of all subtrees, also adding one for itself.

The operator `evaluate` is used by service type `Evaluate`. It computes a preference value from `UpMetric` and `LocalChildMetric` of a `jcq` message. Locally available information is additionally considered. The preference value is expressed as an integer number which is included in the `jca` message that is sent if the preference is acceptable. Whether a preference value is acceptable is determined by a comparison with `PREF_THRESHOLD` which is also defined in package `Metrics`.

The operator `makeDownMetric` is used in service type `Aggregate` to convert the aggregated information from all children, present as `UpMetric`, into a `DownMetric`. This functionality is needed when the node is a cluster head, and the cluster-wide aggregated information is propagated to all members of the cluster in `down` messages.

The operator `downIncrement` performs the necessary modifications to `DownMetric` values before they are resent to the next nodes outward from the cluster head, e. g. incrementing the count of hops from the cluster head.

The operator `clusterTooSmall` is used in a cluster head to determine whether the cluster quality is still satisfactory, e. g. the cluster is not too small to stay independent. The operator is used in service type `Cluster`. It assesses the cluster head's set of direct children and the cluster tree structure. If the cluster is found not to be good enough to persist independently, a cluster head tries and joins another cluster with its old cluster as a new subtree.

2.3.3 Examples

Some examples of metrics that may be specified and used for the assessment of the cluster structure are given in Table 2.1 and Table 2.2. In the left column, the metrics as measured at a single node are given. The expression

local observation $L(n')$ or $L(n)$ if dependent on child node n	up metric in node n: $U(n)$	aggregated value for parent node n': $U(n')$	global down metric U
$L(n) = 1$ if a child is present	depth of subtree	$L(n) + \max U(n)$	max hop distance between node and cluster head
$L(n') = 1$	# nodes in subtree	$L(n') + \sum U(n)$	# nodes in cluster
$L(n') =$ estimated remaining operation time from battery power	min remaining operation time of subtree	$\min(L(n'), U(n))$	min remaining operation time of cluster members
$L(n) =$ wireless transmission quality to child n; $L(n) \in [0, 1]$	lowest communication quality in subtree	$\min(L(n) \cdot U(n))$	lowest communication quality from member to cluster head
$L(n') =$ # neighbours within same cluster except parent and children	intra-cluster links unused by tree	$L(n') + \sum U(n)$	$2 \cdot$ # additional links between nodes of the cluster

Table 2.1: Possible metrics

local observation $L(n')$ or $L(n)$ if dependent on child node n	up metric in node n: $U(n)$	aggregated value for parent node n': $U(n')$	global down metric U
$L(n')$ = set of cluster IDs of neighbouring nodes of other clusters and one gateway node for each cluster	set of other clusters reachable from subtree and their gateways	$L(n') \cup \bigcup U(n)$	set of clusters neighbouring the cluster with gateways
$L(n')$ = # links to nodes of other cluster	# links from subtree out of cluster	$L(n') + \sum U(n)$	# links out of cluster
$L(n')$ = direction and speed of motion, e. g. from GPS	motion similarity degree	polar coordinate interval of $L(n'), U(n)$ or other enclosing area	mobility / cluster cohesion
$L(n)$ = change of transmission quality to node n within last time period; $L(n) \in [-1, 1]$	max link quality change in subtree	$\max(\lvert L(n)\rvert, U(n))$	mobility: max link quality change in cluster
$L(n')$ = # parent losses within last time period	sum of all parent losses in subtree in last time period / divide by # nodes in subtree for meaningful values	$L(n') + \sum U(n)$	stability: # parent losses divided by total number of nodes

Table 2.2: More possible metrics

$L(n')$ is used for a value that is locally measured at the node n' and relates only to the node itself, and $L(n)$ is used for a value that is measured at the node, but that relates to a specific child n of the node. In the second column, the meaning of the aggregated values $U(n)$ received in up messages from the children is stated, while the third column gives the calculation rule for deriving this node's up metric value from the children's values $U(n)$ and the node's own value $L(n)$ or $L(n')$. The last column states the meaning of the value that is aggregated for the complete cluster. This value is computed at the cluster head and may be propagated to all cluster members with the down messages if this is desired.

2.4 Properties

When devising a new clustering scheme for mobile ad hoc networks, care must be taken to ensure that it yields results, i. e. that it terminates when the topology is stable, that it establishes suitable clusters in case of node mobility, and that the resulting structure satisfies the conditions of a clustering structure (depending on the definition used). In the following sections, the properties of ReBaC2 clustering when applied to a stable topology are discussed.

2.4.1 Partitioning

The clustering scheme constructs a partitioning of the given network. This is achieved by the fact that every node has exactly one cluster ID at a time, and the nodes with a common cluster ID belong to the same cluster. If a node loses contact with its cluster and thus ceases to belong to its cluster, it either becomes a member of another cluster or declares itself a cluster head, using its own ID as its cluster ID.

2.4.2 Convergence

As the behaviour of the clustering scheme is repair based, it is not obvious at first sight that the repair actions lead to a stable clustering structure in a stable topology. However, a proof can be given for the case that clusters are not artificially split, and all nodes taking part in the network are switched on simultaneously.

Consider h, the total number of cluster heads in the network. In the beginning, $h = n$, the total number of nodes in the network, because every node

declares itself a cluster head when it is started. When a cluster head decides that the quality of its cluster is not sufficient anymore, it tries to join another cluster with its complete subtree. If this action is successful, the old cluster merges with the new one, and the total number of cluster heads h decreases by one. If no join action takes place, the old cluster head stays cluster head and keeps its members in the cluster. Thus, in this case, the total number of cluster heads h stays unchanged. This means that with every action, the number of cluster heads h either decreases or stays unchanged, where h stays repeatedly unchanged only in cases where there is no change in cluster structure, implying that there cannot be more cluster structure changes than total nodes in the network. This means that, in a stable topolology, the cluster structure formation will definitely reach a convergent state in finite time.

The constraint of a stable topology can be loosened to a constraint only ruling out mobility of the nodes. In this case, nodes may still appear in or disappear from the network at any time. When a new node appears, it will declare itself a cluster head, increasing h by one. When a node disappears, its former cluster will continue to exist if the disappearing node was not a cluster head. The children of the disappearing node will try and rejoin another cluster. If they are successful, the former children will join the new cluster with their complete subtree. If a join operation is not possible for a child, the former child will declare itself a cluster head, increasing h by one. As the number of children of a disappearing node is not generally bounded, no statement can be made regarding the increase of h due to a node's disappearance. Nevertheless, it can be said that if the number of node disappearances is finite, then the number of cluster structure changes is finite as well, and after each node disappearance, a convergent state will be reached after a finite number of structure changes.

A similar argument can be derived for the case of node mobility. A node that moves away from some nodes, and into the reach of other nodes essentially has the same effect as if the node had disappeared from its previous position and reappeared at its new position. If the node movement is broken down into such elementary topological changes, the argument above can be used to show that a convergent state can be reached some time after the node movement.

Chapter 3

Cluster Based Routing

3.1 Introduction

In this work, a hierarchical routing scheme consisting of ReBaC2 and an adaptation of AODV routing, AODVlight, has been composed. As the clustering scheme aggregates intra-cluster routing information, it can be seen as a proactive routing mechanism working in a limited area, while AODVlight, on the other hand, supplies the joint mechanism with the reactive functionality needed to respond to all routing requests that cannot be handled by ReBaC2. AODVlight has been changed to work on an overlay network constructed by ReBaC2, thus reducing the number of messages necessary for AODVlight to search the complete network.

This chapter introduces the hierarchical routing mechanism that has been constructed with the help of ReBaC2. A hierarchical routing mechanism is desirable in ad hoc networks with a large number of nodes, because it reduces the complexity of the network for the single nodes. Without a hierarchical structure, information about the complete network needs to be stored either at every node or at some explicitly selected central nodes. If the network is too large, this is a large storage overhead and many management messages are necessary to keep the stored information up to date. With a hierarchical approach, the network information can be divided and allocated to different nodes.

The hierarchical structure supplied by ReBaC2 is used to alleviate the need for large amounts of routing information in specific points of the network. With the network being partitioned into clusters, only cluster-wide information is collected and stored at centralised points of each cluster. The cluster

heads provided by ReBaC2 have been chosen as the central points of routing information aggregation. Thus, the routing structure of every cluster is stored centrally in the cluster head. This solves routing problems that occur wholely within the cluster.

For routing problems that occur between different clusters, some inter-cluster communication is necessary. To this end, if two neighbouring nodes belong to different clusters, they are regarded as *gateways*, i. e. they provide the capability of the two cluster heads to communicate. For each of these so-called neighbouring clusters, a pair of gateways is stored at both cluster heads in order for each cluster head to be able to communicate with all cluster heads of neighbouring clusters.

It seems reasonable to choose a well-known routing scheme for ad hoc networks to solve the problem of route discovery between multiple clusters, as these routing schemes have been designed for the kind of topological uncertainty that is unavoidable with mobile ad hoc networks. With a proactive routing scheme, in a large network, a large amount of routing information needs to be stored although it is not certain that this information will be needed. Furthermore, with high mobility, the information may be already outdated when it is needed. The reactive approach to routing seems more suitable to a network of arbitrary size, as there is no need to store all available information, but only information on paths that are actually used. As the routing information is established on-demand, it is more likely to be accurate at the time of use.

Choosing a reactive routing scheme for the communication between the clusters that have been formed by ReBaC2 still leaves some questions open. The routing scheme needs to be adapted to be able to take advantage of the hierarchical structure provided by ReBaC2. The approach taken here is to use a reactive routing scheme on the higher level of abstraction, i. e. on a cluster basis rather than a node basis. With the structural information present at the cluster heads, the higher-level routing scheme can be employed to establish routes between cluster heads, while the lower-level, i. e. intra-cluster, routing problems can be solved by ReBaC2.

3.2 Routing Framework

In order to be able to compose different routing schemes to form a new one, the routing framework for mobile ad hoc networks devised in [FGG07] is used. The block structure of its SDL specification can be seen in Fig. 3.1. The block `routingMiddleware` consists of a packet forwarder and a routing

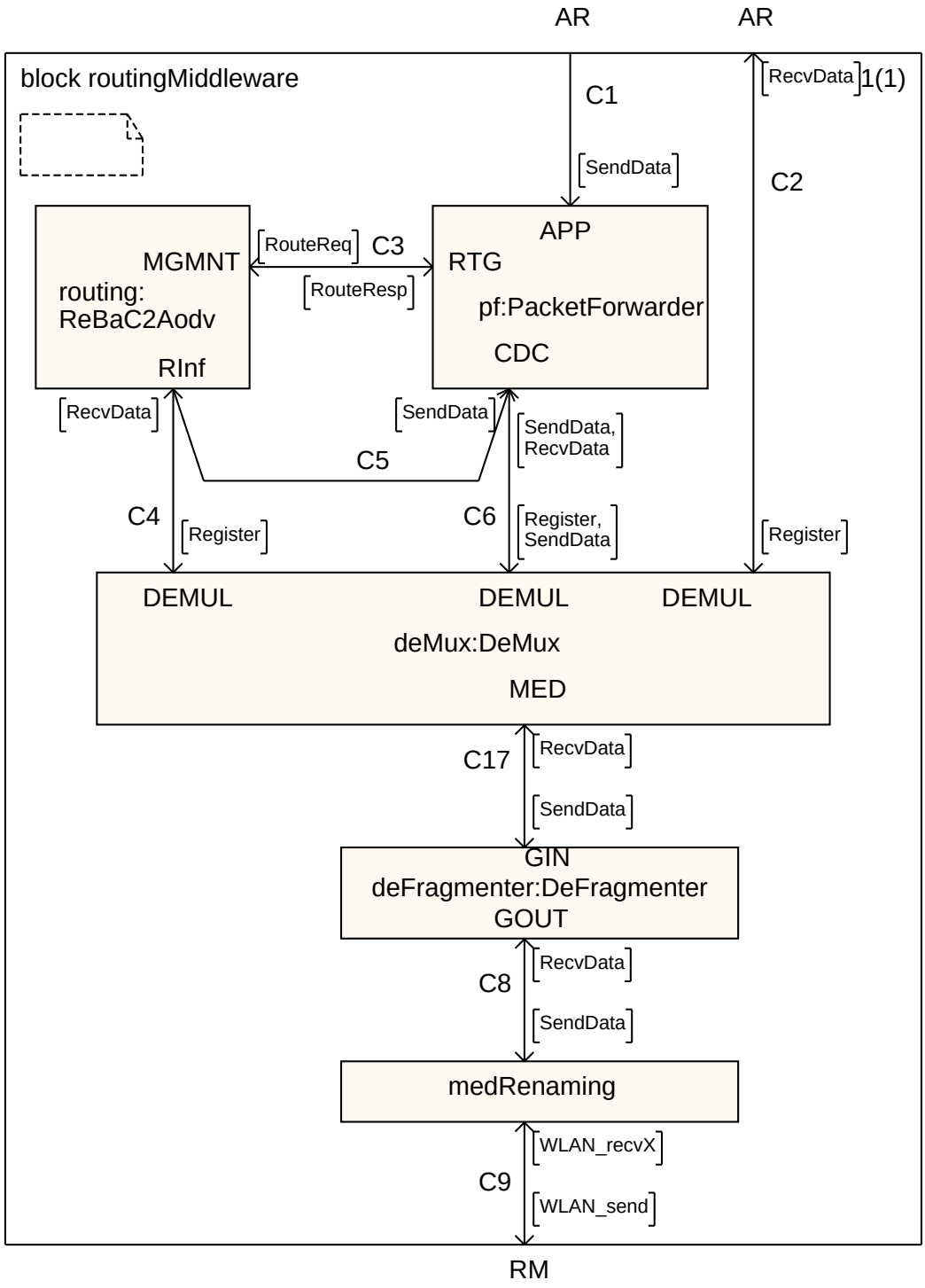

Figure 3.1: Block `routingMiddleware`

scheme. The packet forwarder forwards incoming packets to other nodes in the network if required and obtains the necessary routing information from the block `routing`.

The framework has been designed in order to make a large variety of different approaches to routing possible. This is achieved by defining a simple interface and making the routing mechanism exchangeable. The routing functionality can be defined by specifying a block type that is then instantiated as block `routing`.

When a message intended for another node is examined by the `PacketForwarder`, a RouteReq signal is sent to the routing component. The routing component replies to each RouteReq signal with a RouteResp signal informing the `PacketForwarder` of the path the given packet should take.

The RouteReq and RouteResp signals use a data type called `Path`. It can represent unicast and multicast paths in a network. In the `Path` type, there are different types of links between nodes, most notably direct links and standard links. A *direct link* represents an actual link between two nodes on the physical layer. A *standard link* represents the fact that the two nodes should be able to be connected with a sequence of direct links across other hops, but that the exact path information is currently not known. The combination of these two link types makes it possible to provide a `Path` from source to destination, even if the actual network path is only partially known. The part that is known is given in the form of direct links, and the unknown part is left for further elaboration in the form of a standard link.

The nodes in the `Path` type can have different properties. One of these properties is the *search* flag. If this is set for a node, it means that the path to this node is being looked for at the moment. So, a typical RouteReq signal contains a `Path` from source to destination, where a part of the path consists of a standard link, and the node that the standard link leads to is marked with the search flag. This means that the routing scheme that receives the RouteReq signal should try to replace the standard link to the node in question by a sequence of direct links, or at least by some direct links and a standard link, giving part of the information that is required. If the routing scheme does not have sufficient information to replace the standard link in question, it leaves the `Path` unchanged. This makes it easy to use the same `Path` in consecutive queries to different routing mechanisms. Each mechanism adds the information it has to the `Path` and returns the modified `Path` with the RouteResp signal. This path can then directly be used in another RouteReq signal to another routing mechanism.

The `Path` type explained here makes is possible to build the path to a destination node on the way. The traffic is not completely source routed, but it is not only routed by the single hops it visits either. It is merely a combination of the two. Portions of the path can be routed similar to source routing by specifying a sequence of direct links in the path. At other points in the path, the decision about the next hop from a certain node can be left open until the message reaches that node, and that node can then provide one hop, a part of the path, or the complete path, depending on its knowledge. This feature is well suited for the hierarchical routing that is established in this work, because the part of the route that lies within one cluster can be supplied by one node, while the parts of the route that lie within different clusters can be left vague until the message arrives there. It is even possible to specify a path on the hierarchically abstracted view of the network by providing a sequence of standard links between nodes of different clusters, each of which will be replaced by a sequence of direct links at each cluster.

Furthermore, the communication protocol that is used in the framework supports the use of hierarchical application-specific addressing. With the help of this feature, the routing functionality can be further split into sub-mechanisms each using its own address space, connected within block `routing` by multiplexing agents. In this way, new routing schemes can be easily devised as compositions of other mechanisms.

The complete SDL system can be seen in Fig. 3.2. It consists of an example application that can regularly send data to other nodes, and block `routingMiddleware`, which contains the forwarding and routing functionality as shown in Fig. 3.1.

3.3 Division of Activities

Using two different routing protocols at the same time makes a clear division of responsibilities between the two necessary. When a routing problem emerges, it is desirable to make use of the two different protocols in a way that saves resources.

For example, if there is a proactive scheme, it can be queried without incurring a penalty in terms of messages. On the other hand, the proactive scheme will probably have been configured in a way that it cannot answer queries relating to the complete network, or else another routing scheme would not be required. So, a query to the proactive scheme may be met with an invalid answer if the knowledge to answer this query is not present.

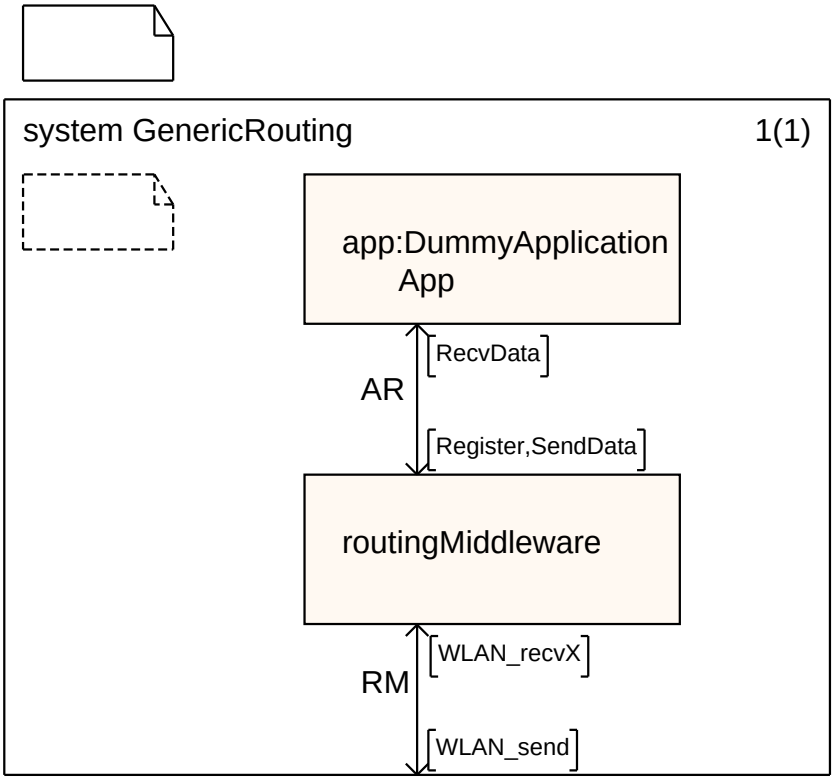

Figure 3.2: SDL system diagram

In this work, it has been decided to use ReBaC2, the existing clustering algorithm, as a proactive scheme confined to a certain locality. All queries reaching outside of that locality are processed by a reactive scheme. Additionally, the reactive scheme being used for the network-wide queries should be able to benefit from the hierarchical structure provided by ReBaC2. This combination is supposed to make the reactive scheme more efficient by its usage of the proactively established cluster-wide information. The division of responsibilities between the two routing schemes has been made depending on the locality of the routing problem. For routing problems that occur within a subtree of the same cluster, ReBaC2 is responsible without additional involvement of the other scheme, and for all other routing requests, a reactive routing scheme is used.

For the other routing component besides ReBaC2, a reactive routing scheme is preferred. This is due to the fact that reactive schemes do not need to collect information on the complete network in advance. Thus, using a reactive scheme makes the message and storage complexity of the establishment of the basic network structure independent of the actual size of the complete

network. (Additional messages are of course needed once specific routes are requested from the mechanism.) ReBaC2 as a locally confined proactive routing scheme is dependent in its complexity only on the maximum size of a cluster, so its complexity can be independent of the total network size if the cluster formation metrics are adjusted accordingly.

A further requirement to the second routing mechanism is that it should be able to take advantage of the hierarchical structure provided by ReBaC2. This can for example be achieved by using an overlay network structure that is inferred from the clustering structure, e. g. associating each cluster built by ReBaC with one node in the overlay network. The routing scheme should thus be easily adaptable to be used in an overlay network.

3.4 Choice of Routing Mechanism

The routing protocol was chosen with the following considerations. The cluster structure should be used to to create a hierarchical routing scheme. In order to achieve the level of abstraction necessary for hierarchical structuring, an overlay network is defined. This abstract overlay network is implied by the cluster structure on the underlying network.

The routing mechanism used on the abstract overlay network needs to be able to cope with typical mobility related issues, such as unpredictable topology change, because these properties of the underlying network cannot be hidden from the overlay network. Furthermore, a reactive routing scheme is desirable in order to save bandwidth at times when no new routes need to be discovered.

For the above reasons, AODVlight (see Section 1.2.2) was chosen as the routing scheme to be applied to the overlay network generated by ReBaC2. In order to build a routing scheme on the existing clustering structure, AODVlight has been adapted to form a hierarchical routing mechanism using the clustering structure to provide the abstraction from the underlying network.

The cluster structure is used to generate a dynamic overlay network depending on the pair of source and receiving node. A node of the underlying network is a node in the overlay network if it is a cluster head or the source or destination of the message that is sent. There is a link in the overlay network between two cluster heads if there exists a pair of gateway nodes each of which belongs to one of the clusters of the cluster heads being connected, and the two gateway nodes are direct neighbours in the underlying network. There is also a link in the overlay network between a cluster member and its cluster head.

The routing information that is necessary to deliver data along a link of the overlay network is locally present as a part of the cluster structure information in the nodes.

3.5 Coordination of ReBaC2 and AODVlight

In block type `ReBaC2Aodv`, the composition of the two aforementioned mechanisms is specified. The design of the new routing scheme is influenced by the framework being used, and by the clear separation between ReBaC2 and AODVlight that is to be preserved. Thus, the two mechanisms appear as separate blocks connected only via manager and adapter blocks (see block `type ReBaC2Aodv` in Fig. 3.3).

The adapter block `ReBaC2AodvAdapter` is a bidirectional multiplexer. The combination of both mechanisms is viewed as a single source for routing information by the packet forwarder. This simplification is achieved by a central management component, the block `ReBaC2AodvManager`. It directs incoming RouteReq signals to both components and consolidates the information yielded from these queries.

3.5.1 Behaviour of AODVlight

Resulting from the above mentioned generation of an overlay network is the following behaviour of the employed AODVlight scheme. AODVlight works only on the overlay network, i. e. on a part of the network consisting of source node, destination node, and all cluster heads. AODVlight is used to reactively find a path on the overlay network. To this end, the broadcast mechanism that AODVlight relies on is changed such that it no longer sends local broadcasts but sends messages that are broadcasts on the overlay network. Thus, far fewer nodes participate in the AODVlight route discovery process than if pure AODVlight was used without clustering. This also means that AODVlight routing information is only kept at the source and destination nodes of a route and at intermediate cluster heads.

The management behaviour that is used to resolve route requests to the combined routing scheme is shown in Fig. 3.4. When a RouteReq signal is sent to block `ReBaC2AodvManager`, process `InstantiatorReBaC2Aodv` creates a new instance of process `ReBaC2AodvManager`, which processes exactly one RouteReq signal. This process defines the order in which the route requests are handled by the two routing schemes. Currently, an incoming route request

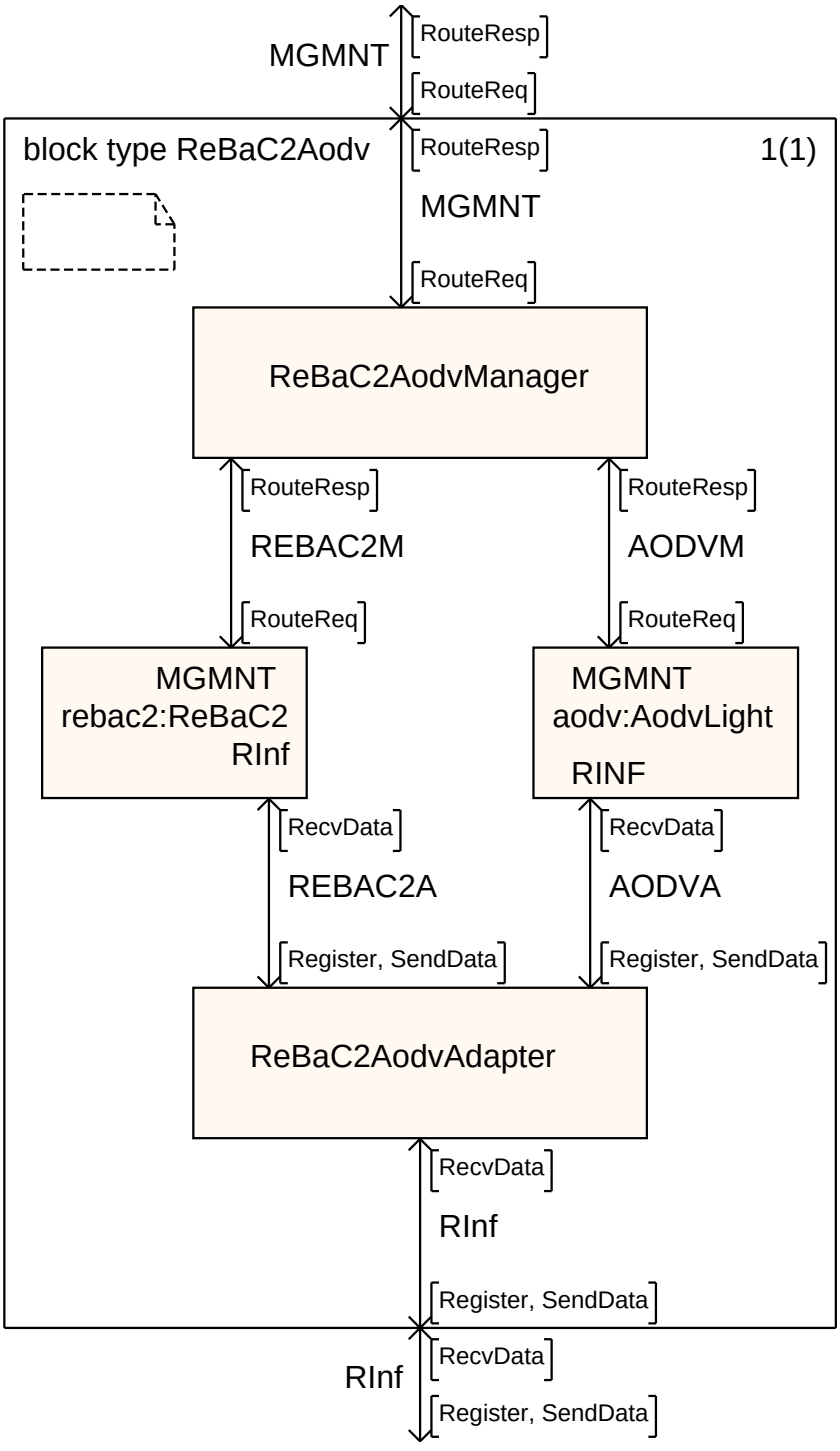

Figure 3.3: Block type `ReBaC2Aodv`

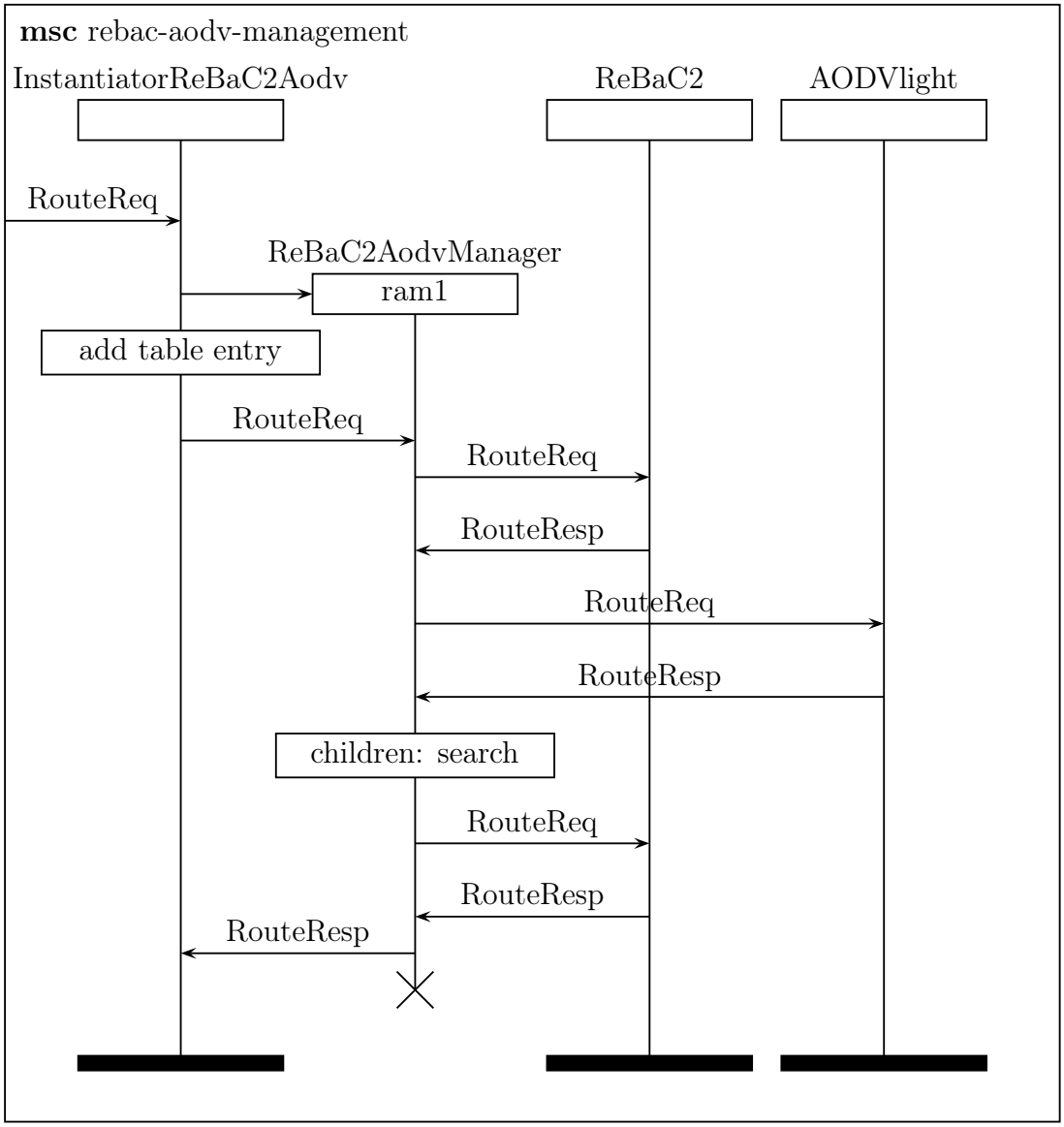

Figure 3.4: Processing of routing queries in the management of ReBaC2 and AODVlight

is first sent to ReBaC2. ReBaC2 answers it with as much information as it has present, and leaves the rest of the query unanswered. The path returned by ReBaC2 is then sent to AODVlight in a new RouteReq signal, so that AODVlight tries to answer the queries remaining from ReBaC2. AODVlight uses its route discovery process to find out which of the neighbouring cluster heads is the next hop on the overlay network towards the destination. In the resulting answer, all nodes that are direct children of the processing node are marked `search`, and the request is again sent to ReBaC2 for processing. This step is necessary because the result from AODVlight may contain nodes as next hops that are next hops in the overlay network, but not actually in the underlying network. Thus, these overlay next hops nodes have to be looked up by ReBaC2 in order to make the resulting path feasible on the underlying network.

In this sequence of route requests, each process receiving a route request adds the relevant information that it has to the path, and the complete path with the newly added information is then sent back to process `ReBaC2AodvManager` with a RouteResp signal. Thus, with the given sequence of route requests, the information in the path is incrementally completed with the knowledge of both routing schemes. ReBaC2 is considered first because it is a proactive scheme, meaning that no communication overhead is incurred by requesting information from it, no matter whether the requested information is available or not. After that, when the route request is sent to AODVlight, its reactive route discovery is only used if the path still lacks information on the next hop towards the destination. In the end, the proactive ReBaC2 is used to fill in the local information that may have been left out by AODVlight because of its operation in the overlay network.

3.5.2 Routing Query Processing by ReBaC2

An example of a routing query being processed by ReBaC2 components is shown in Fig. 3.5. The service `rou` is the instance of service type `Route` in process `ControlServices` (see Fig. 2.13), and process `management` is the only process in block `ReBaC2Management` (see Fig. 2.7). The process `management` is the component that answers incoming RouteReq signals with the help of the cluster-based information available from service type `Route`. The requests are answered by adding all available information to the path received with the request and returning the modified path with a RouteResp signal.

When the RouteReq signal is received by process `management`, it searches the given path for the address to be looked up, and finds the special address `ADDR_ALLCH`, which means that the incoming query requires information for

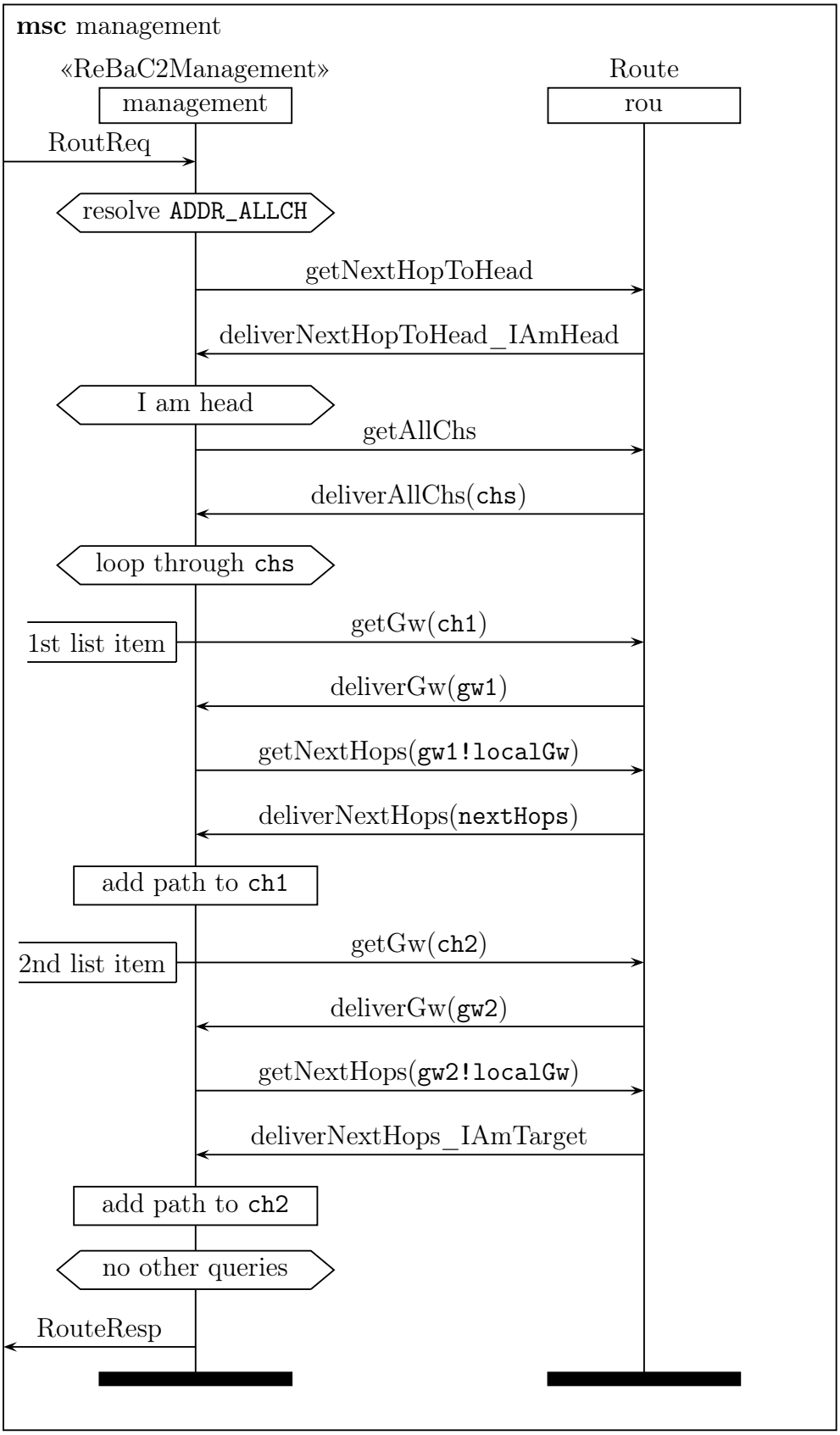

Figure 3.5: Processing of routing queries in ReBaC2 management

broadcasting a message, e. g. an RREQ message, on the overlay network, i. e. to all neighbouring cluster heads. To reach all neighbouring cluster heads, a message first needs to be delivered to the local cluster head. This is why the first request sent to service `rou` requests the next hop towards the local cluster head. The answer to this request shows process `management` that this node is currently a cluster head. This means that the information on the neighbouring cluster heads is present at this node. The signal getAllChs requests the list of neighbouring cluster heads. Assume that this list includes two cluster heads, namely `ch1` and `ch2`. For each of these cluster heads, the gateway towards its cluster is requested from `rou`. A gateway is represented by the data type `ClusterGateway`, which includes the gateway node in the local cluster (`localGw`), the gateway node in the remote cluster (`remoteGw`), and the remote cluster ID (`remoteCid`).

From the gateway information of `gw1`, the local gateway node ID is taken and looked up at `rou`. The information on the path towards the local gateway is present at `rou` because it involves only intra-cluster information. For `ch1`, the hops from this node towards the local gateway are added to the path as direct links, then the direct link from to local gateway to the remote gateway is added, and a standard link from the remote gateway to its cluster head. The standard link can be replaced by explicit direct links at the remote gateway node, because it is a member of the target cluster and thus the information on the path towards its head is present there.

For the second cluster head in the list, the gateway is also requested, and the path to the local gateway is looked up. In this case, the local gateway towards `ch2` is identical to this node itself. Thus, the resulting path consists of a direct link from this node to the remote gateway, and a standard link from the remote gateway to its cluster head. As there are no other queries, i. e. nodes marked as `search`, in the path received with the RouteReq signal, the processing of the query has been completed, and the path with the added information is sent back to the requesting process in a RouteResp signal.

3.6 Collaboration

This section describes the behaviour of the resulting distributed algorithm. First, the behaviour of ReBaC2 is presented, and second, the function of the routing protocol resulting from the combination of ReBaC2 and AODVlight is detailed. For an example of a network structure with clusters and gateways see Fig. 3.6. In this diagram, the blue nodes are cluster heads, and the other nodes are cluster members. The two nodes that share the link between the

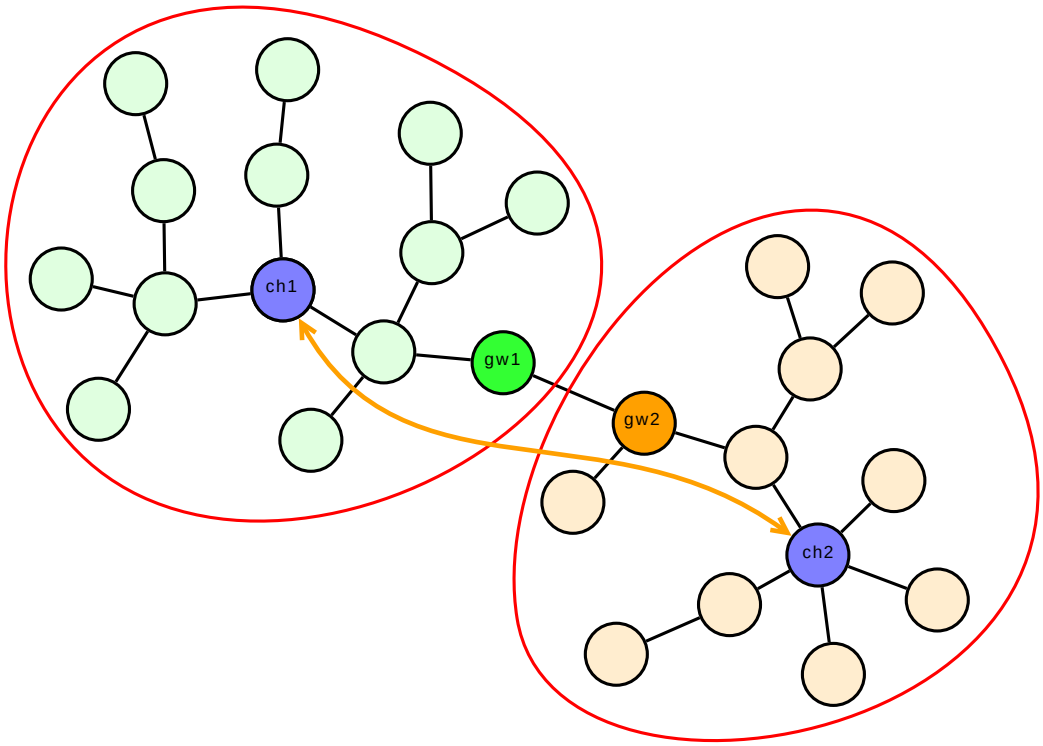

Figure 3.6: Two clusters linked by gateways

two clusters are gateways. The link of the resulting overlay network is shown as an orange bent line with arrowheads.

The transformation of the underlying network to the overlay network is described in Section 3.4. It is achieved by the clustering scheme in the following way. The clustering structure is constructed as described in Section 2.2.2. The cluster heads have knowledge of all members of their cluster, and of the complete paths towards all members of their cluster. Additionally, for each neighbouring cluster, they know which of their members has a direct neighbour that belongs to the neighbouring cluster. For each of these local gateways (gateways that are in this cluster), the remote gateway (gateway in the remote cluster) and cluster ID (ID of the remote cluster head) is also known. This makes it possible to explicitly construct the route up to the remote gateway at the cluster head, so that the message can be delivered into the remote cluster with the knowledge of the sending (or forwarding) cluster head. From there, the intra-cluster route towards the cluster head can be used.

Every node that is a member of a cluster knows the next hop towards its cluster head. As the link structure within a cluster is that of a tree rooted

in the cluster head, every node has the possibility to send a message to its cluster head, even if it does not know the complete path.

With the knowledge present in the clustering structure, it is possible to establish a path from one cluster head to another neighbouring cluster head without acquiring additional information. This capability is used by AODVlight when addressing neighbouring cluster heads. To achieve this, the special address `ADDR_ALLCH` has been introduced that denotes the local cluster head at a member node or all neighbouring cluster heads at a cluster head. It can thus be seen as the equivalent of a local broadcast in the overlay network. This address is used in AODVlight as destination of the otherwise locally broadcast RREQ messages. A message addressed to `ADDR_ALLCH` is first delivered to the local cluster head. There, it is explicitly addressed to all remote gateways of neighbouring clusters. As the paths to the remote gateways are completely known at the cluster head, they are included in the message. From the remote gateways onwards, the message is vaguely addressed to the then local cluster head. The remote gateway is itself a member of its cluster and thus has the capability to deliver the message towards its cluster head. A receiving cluster head treats incoming RREQ messages as locally broadcast flooding messages and either replies to them or resends them on first receipt towards its neighbouring cluster heads.

In more detail, the following happens. When the path to a destination is not known, the node wishing to send a message starts a RREQ message. It is addressed as a local broadcast in the overlay network. This is achieved by addressing it to the special address `ADDR_ALLCH`. The underlying network delivers these messages to all neighbouring nodes of the overlay network, i. e. to all neighbouring cluster heads and to the destination if it is available. Two cluster heads are said to be neighbouring when they are linked in the overlay network. Whether the destination is available at a cluster head can be found out by querying ReBaC2 about the destination node. As the cluster head is the root of the tree that connects all cluster members, it knows all cluster members and can answer the question whether the destination is in this cluster. First, AODVlight tries to send the message only to the destination node itself. If this succeeds because the destination is known to ReBaC2, the destination node receives the RREQ message and answers it. The answer is a RREP message that is sent along the path of cluster heads towards the cluster head of the source's cluster and then the original source itself. When the overlay broadcast message is received at a cluster head where the destination is not directly available, the message is rebroadcast to all neighbouring cluster heads.

When a RREP message is delivered backwards, it is not addressed to all neighbouring cluster heads, but explicitly to the intended neighbouring cluster head. Together with the information on local and remote gateway nodes, the cluster head also knows the remote cluster head's ID, and can thus choose the correct pair of gateway nodes to forward the message to.

When the RREP message is received at the node that originally started the request, the route is completely set up. The path from the source to its cluster head consists of next hop information towards the cluster head. The path from the source's cluster head to the destination's cluster head possibly contains other cluster heads. Each of these head-to-head links consists of the explicitly addressed path from the sending cluster head to the particular remote gateway node, and the next hop information from the gateway node to its cluster head. At the destination's cluster head, the path information on cluster members is used to explicitly address the message up to the destination node.

Messages in the opposite direction can be delivered in a similar manner, but a completely new route request is necessary, because AODVlight does not set up the reverse path when establishing a link. However, the links of the overlay network are bidirectional because the gateways in ReBaC2 always inform both cluster heads of their existence. Thus, a route for the opposite direction can be found by AODVlight if the original route has been established.

3.7 Conclusion

The combined routing scheme that has been proposed is a hybrid routing scheme consisting of a proactive and a reactive part. The proactive part is used for a limited area around the nodes, the clusters, and the reactive part is used when the routing problems reach beyond this limited area. The advantages of the proposed combination of routing schemes are summarised in the following paragraphs.

A hierarchical abstraction of the underlying network is constructed. The proactive part of the scheme is used for all routing requests that can be answered within the network parts defined by the hierarchical structure, i. e. the clusters. This means that for such a route request, no additional route discovery process is needed.

For route requests that reach outside the clusters, the reactive functionality is used. This has the advantage that a reactive route discovery is not needed

in all cases but only in the cases where several clusters are affected by the route. Additionally, the reactive routing scheme uses the overlay network implied by the hierarchical structure in order to more efficiently search the complete network. By using an abstraction of the actual network, the reactive scheme needs to cope with far fewer nodes, thus reducing the total number of messages and the memory requirements for the nodes of the reactive scheme.

Chapter 4

Simulations

The SDL specification of the algorithm was transformed to *C++* code using the *Configurable SDL Transpiler and Runtime Environment* (ConTraST) [FGW06]. The behaviour of the resulting executable was simulated with *ns+SDL* [KGGR05], an extension of the network simulator *ns2* [ns2].

Simulations have been conducted with the following parameters:

- 56 nodes, randomly placed on an area of 400·400 metres, no node mobility
- Wireless communication radius of each node: 73 metres
- Parameter to ReBaC2: minimum cluster size set to 4, i. e. a cluster should have at least 3 members apart from the cluster head.

The simulation results of one of these scenarios are depicted in the following diagrams. In these diagrams, a red square denotes a node that is a cluster head and a black plus sign shows a normal node. A black, solid line shows a link that is part of the cluster structure, i. e. that links two members of the same cluster, and a green, dotted line shows a link between two gateway nodes of different clusters.

In this simulation, all nodes are switched on at the beginning. After being switched on, all nodes declare themselves cluster heads and wait for member nodes to join their clusters. This state is shown in Fig. 4.1. When the cluster does still not have a sufficient number of members after `noChildrenTimeout`, the cluster heads start trying to become a member of another cluster. In the simulation, the values for `noChildrenTimeout` have been randomly chosen so that the cluster heads do not all start joining other clusters at the same time.

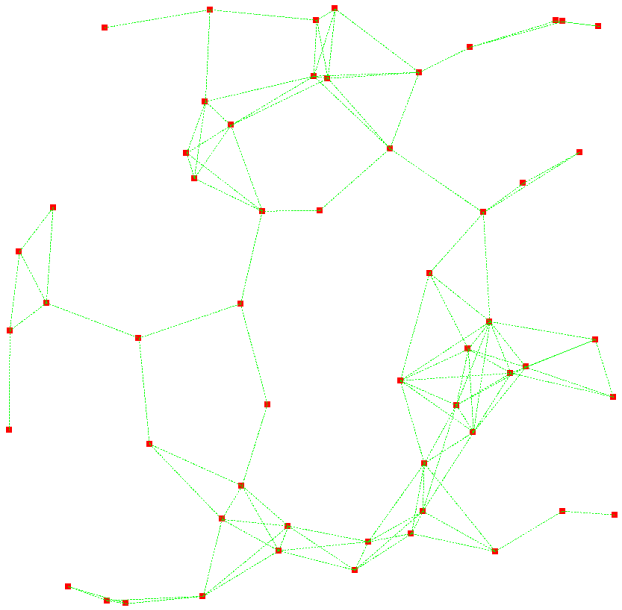

Figure 4.1: Reachability of nodes

Fig. 4.1 also shows the topology of the network that is used in the simulation. There is an edge between all nodes that can directly communicate. The diagram has been generated at the beginning of the simulation, when all nodes are cluster heads, and it shows that every node regards itself as a gateway to all other neighbouring nodes, i. e. clusters.

The cluster structure that has been established by ReBaC2 is shown in Fig. 4.2. Cluster heads are depicted as red squares. Note the cluster in the lower right corner that consists of fewer than four nodes even though it is not completely isolated. This situation persists because the cluster head of the small cluster is not directly reachable by nodes of another cluster. Thus, when it tries to find other clusters to join, it does not find any, and it stays a cluster head with its few members.

Fig. 4.3 shows the cluster structure with the gateways between the clusters. Both nodes at the end of a green dotted line are gateway nodes for their cluster, connecting their cluster to the other cluster they have a link to. There may be several gateway links between a pair of clusters, but in ReBaC2, only one gateway per neighbouring cluster is used at the cluster head. Note that a node can be a gateway to several other clusters at the same time.

In Fig. 4.4, the orange multi-hop arrows show the path that the AODV RREQ message has taken on the overlay network to establish the given route. The

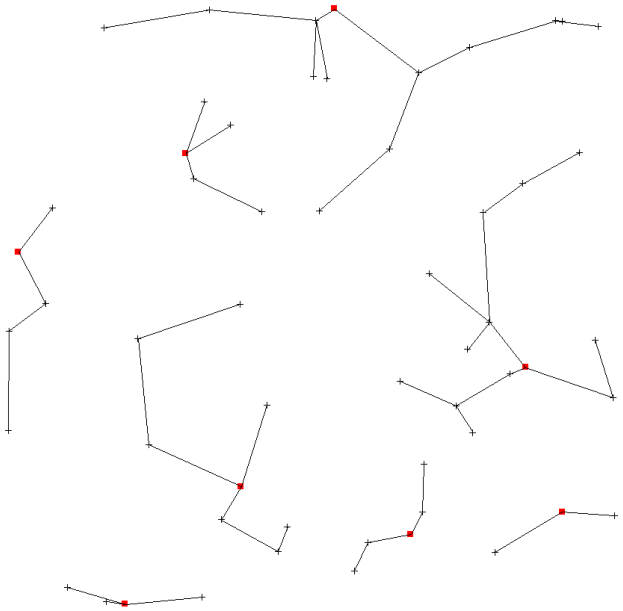

Figure 4.2: Cluster structure

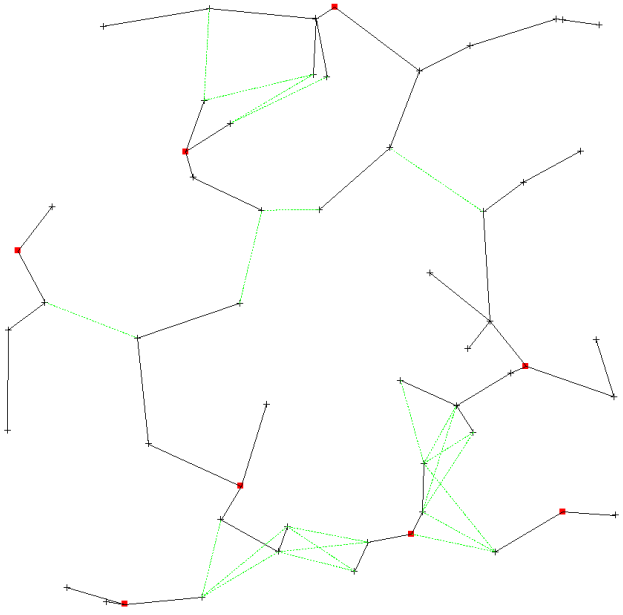

Figure 4.3: Cluster structure with gateway links

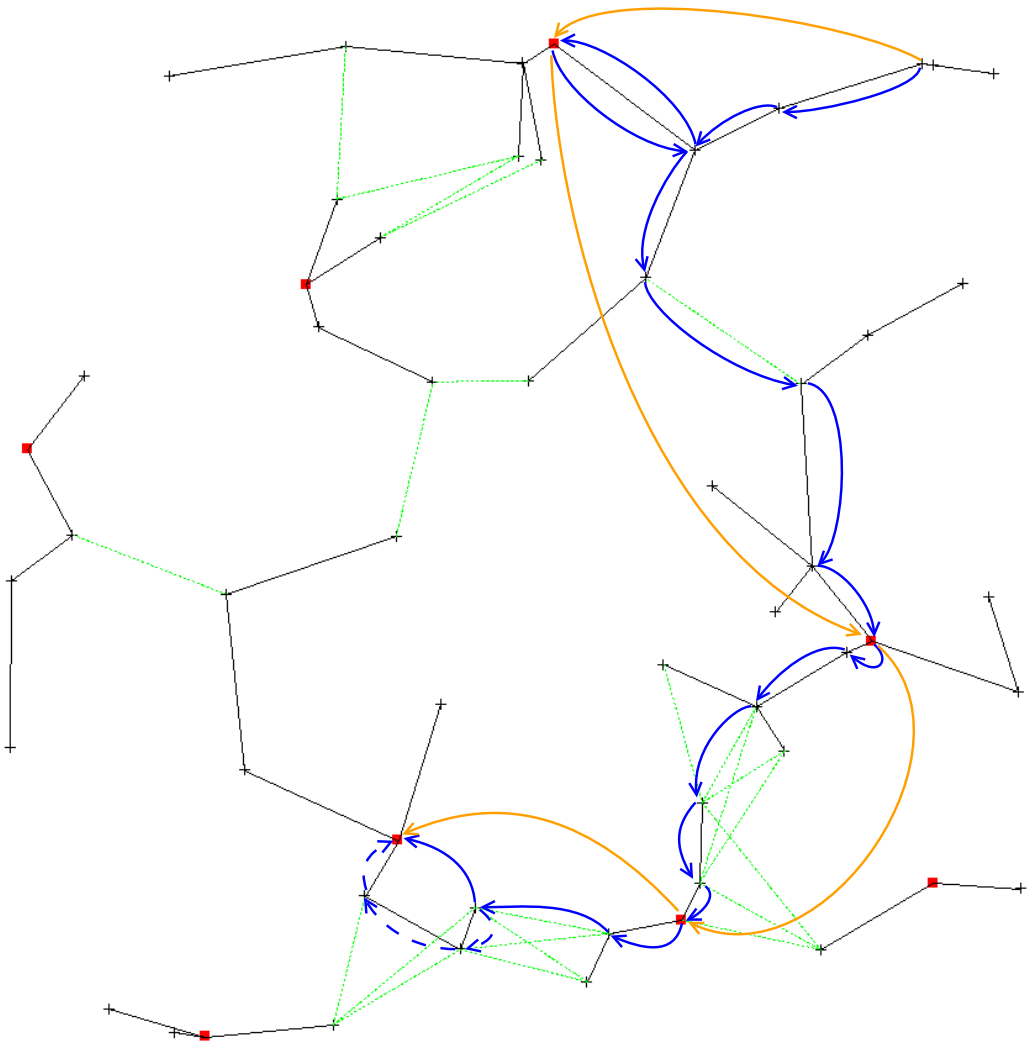

Figure 4.4: Cluster structure with data path

path of an application data message as established by the combined routing scheme is shown with blue arrows. From the source node, the application data message is first sent towards the source's cluster head. From there, it is sent via two intermediate clusters, i. e. via their cluster heads, towards the destination's cluster head. In this case, the destination node itself is a cluster head.

A data message may use the same edge of the network graph twice. In Fig. 4.4, this happens at the edge connecting the source node's subtree to its cluster head. Such a situation occurs whenever the source of the message is in a common subtree with the destination of the message. Source and destination of the message may in this case also be gateways receiving from and sending to other clusters. In the case shown in Fig. 4.4, the source node and the destination gateway are both in the same subtree of the cluster head, thus sharing one hop in their paths to and from the cluster head. As the information on gateways is only used at the cluster head, the message needs to be sent to the cluster head in order to be addressed to the correct gateway. This is, of course, not an optimal routing solution. A possible solution to this problem will be outlined in Section 5.2.2.

Another interesting observation is that the last hop the data message actually takes is not a part of the planned route towards the destination. The planned route uses only links that are part of the cluster structure tree, as depicted by blue, dashed arrows. In this case, when the gateway of the destination's cluster resends the message, the message is directly overheard by the actual destination node of that message. In such a case, the destination node accepts the data message even if it is not on the intended path. The nodes on the intended path have no knowledge about the fact that the message has already been received by the destination and thus also forward the message along the intended path towards the destination.

The above behaviour results in duplicate delivery of the data message, meaning that the medium is not optimally used. While in this special case the problem can be solved by an optimisation of intra-cluster paths as will be described in Section 5.2.1.1, such incidents are still generally possible with the hierarchical network structure constructed by the algorithm.

An example of another such case of duplicate delivery due to suboptimal routing paths is shown in Fig. 4.5. Assume that the message comes from the cluster on the left, and is thus sent from `gw2` to `gw1`. The intended path goes through the cluster head of the destination's cluster, `ch1`, and then to the destination node `dest`. When `gw1` sends the message to the cluster head, `dest` overhears the message and delivers it to its application layer. The cluster head will additionally forward the message because it does not

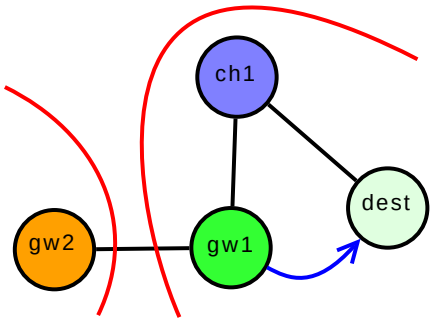

Figure 4.5: Data path shortcut scenario

know that the message has already been received by the destination. This example shows that the problem may exist even though the intra-cluster link structure is simple and contains no indirect links to the cluster head.

Returning to Fig. 4.4, the path that the AODV RREQ message takes to discover the route is shown by orange arrows spanning multiple links. These arrows show the high-level view on the network that AODV uses. The RREQ message originates from the source node and is first sent towards the local cluster head. From there, the message is broadcast to all other neighbouring cluster heads. (In this diagram, only the path of the successful RREQ message is shown.) Once arrived at the destination's cluster head, the RREQ message is unicast to the destination node so that the destination node can generate a RREP message and issue a destination sequence number for it. The abstract path found by AODV together with the detailed information available in ReBaC2 leads to the resulting data path that is shown in the diagram.

Another application data path with different source and destination nodes can be seen in Fig. 4.6. In this example, the delivery of the data message to the destination node is of interest. As shown by the orange multi-hop arrows, AODV finds the route that uses the destination node's cluster head. In this case, the destination node is at the same time the gateway for the neighbouring cluster that the message is received from. The route through the destination cluster as originally intended consists of the destination cluster's gateway, the path to its cluster head, and the path to the destination node. In this case, this would mean traversing the two hops between the destination and its cluster head twice. What actually happens is that the destination node, when receiving the message, identifies the incoming message as addressed to itself, and accepts it. Despite the fact that the path information in the message instructs the node to forward the message towards its cluster head, it does not do so as long as it is the only destination

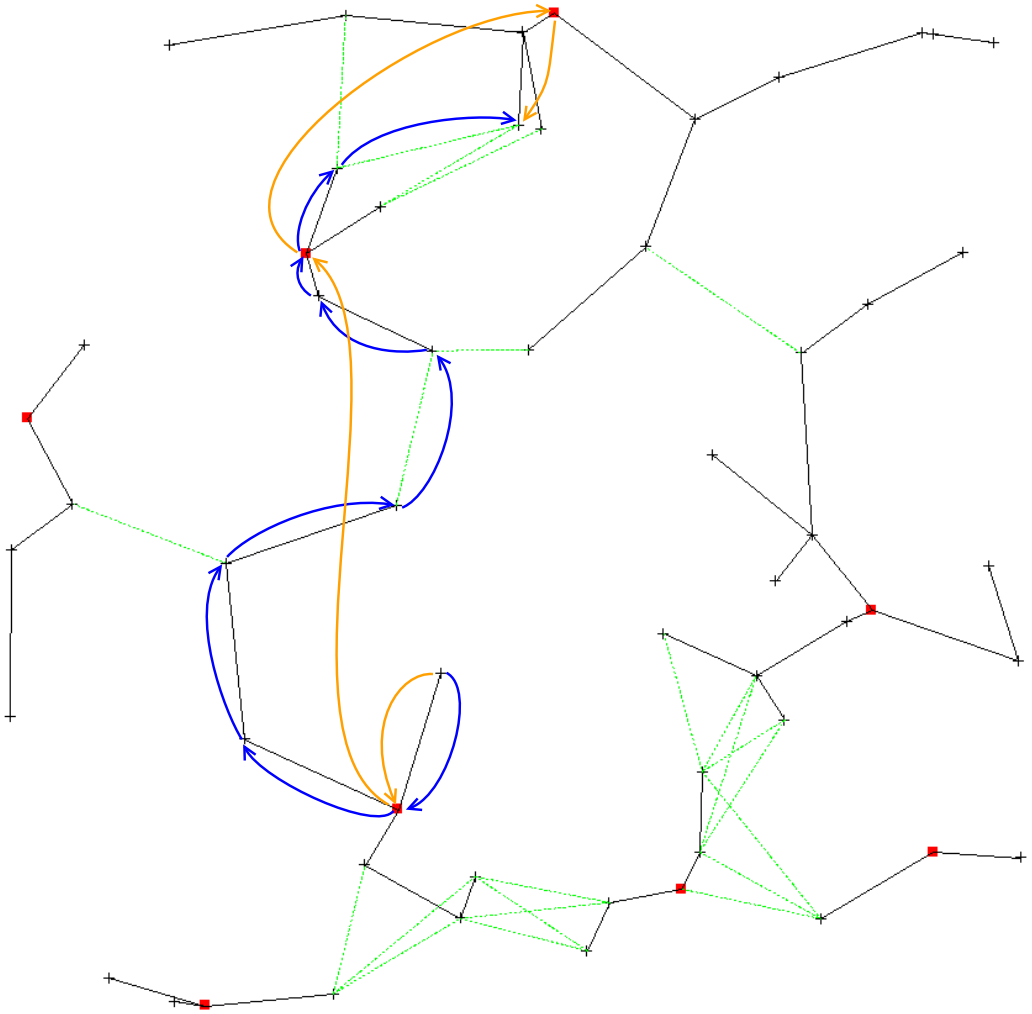

Figure 4.6: Cluster structure with another data path

node of the message. The route that is actually used by the data messages is shown in the diagram by blue arrows.

The simulations have shown that the composition of a hybrid routing scheme using the routing framework provided in [FGG07] is feasible, and that it yields a functional routing scheme while at the same time allowing for the reuse of components such as AODVlight. The resulting algorithm works in different scenarios, some of which have been shown in this chapter. The simulations have also shown that while the route discovery process works reliably, there is some room for improvement of the resulting routes in terms of path length and bandwidth utilisation. Possible improvements to the routing scheme will be discussed in Section 5.2.

Chapter 5

Conclusion and Outlook

5.1 Conclusion

In this thesis, the second version of the repair based clustering algorithm, ReBaC2, has been presented. Its maintenance based approach has been designed to be suitable for mobile scenarios, as this allows for a quick adaption of the cluster structure to topology changes and thus produces a more reliable clustering structure. There have been some enhancements compared to the first version of this algorithm.

One improvement in the second version of the mechanism is the exchangeability of the cluster formation metrics. These metrics can now be specified separately from the algorithm behaviour, thus making it easier to adapt the scheme to specific needs. Another addition is the new support for the use of the clustering algorithm as a small-scale proactive routing scheme for intra-cluster routes. To make ReBaC2 more tolerant of control message loss when maintaining the cluster structure, a dependable message delivery scheme has been specified for some of the clustering message types.

The feasibility of the composition of different routing schemes with the given framework has been demonstrated by specifying a composition of ReBaC2 and AODVlight. The resulting hierarchical hybrid routing scheme has been simulated, and its functionality has been observed in the simulation results. The advantage of the presented hierarchical routing scheme is that by simplifying the view on the network structure for both schemes involved, bandwidth and memory consumption can be reduced in comparison to a flat-routed scheme. As it is based on two algorithms that are suitable for node mobility, the resulting scheme offers good prospects to be suitable for use in highly mobile environments.

5.2 Outlook

Although the basic usability of the composed routing scheme has been shown, there are several points that need improvement before it can be used in productive environments.

5.2.1 ReBaC2

This subsection deals with the possible improvements regarding the cluster structure, cluster formation behaviour, and metrics definition of the clustering algorithm ReBaC2.

5.2.1.1 Intra-Cluster Links

The paths within the clusters constructed by ReBaC2 are not optimal. Currently, there is no check to make sure that the link structure within a cluster is sensible with respect to the reachability of nodes. A node may be linked to an indirect parent node via other nodes despite the fact that it is directly reachable by this parent node.

To counter this problem, a node should be able to change its location in the cluster tree. For example, a node that is linked to the cluster head via another intermediate node may overhear a message directly coming from the cluster head. Based on this observation, the node should be able to switch over to being directly connected to the cluster head. In another case, a node overhearing another cluster member which is closer to the cluster head than the node's parent should be able to switch to the closer node as new parent, thereby shortening its path to its cluster head. If all nodes have these capabilities, the resulting tree will tend to have fewer hops between the cluster head and its nodes, and the nodes will tend to have a higher number of direct children.

5.2.1.2 Cluster Splitting

Clusters currently have the capability to merge, but they do only have the possibility to split up into separate clusters in case of link loss due to topology change. An explicit cluster split functionality can be useful in cases where a cluster has become too big or is otherwise not optimal anymore. Even if it is not for its size, a cluster can be split up, for example if it can be expected that the resulting cluster fragments can merge with different neighbouring clusters to form a structure that is better according to the specified metrics.

This could be implemented by each subtree evaluating its possible other cluster memberships concurrently to its current membership. Each node could periodically send a cluster join request to other clusters in its direct neighbourhood. The replies from the other clusters could be evaluated and compared to the current situation in the current cluster, and if a significant improvement is to be expected, the node could decide to change its cluster membership, taking its subtree with it.

5.2.1.3 Metrics

More, different metrics can be devised, and the metrics can be evaluated more elaborately. Currently, simple metrics have been used to show the feasibility of the exchangeable metrics concept. When the clustering and routing schemes are used in particular application contexts, adequate metrics can be chosen and care can be taken to evaluate the metrics in a way that is beneficial to the needs of the specific application.

This can be done by analysing the properties of the traffic that the application generates, and finding metrics and criteria that cause the given algorithm to generate a network structure suitable for an efficient handling of the application traffic. Testing the devised metrics together with the actual application is advisable in order to be able to further adapt and improve the metrics and criteria.

5.2.2 Routing

The resulting composed routing scheme also needs further improvement because the paths that are used by application data messages are not optimal. This is primarily a result of the hierarchical structure of the network. The messages are delivered from one cluster head to another cluster head via the gateway nodes that are known between these two clusters. As the path information from source to destination is split up into legs from cluster head to cluster head, the resulting path is not at all optimal. For example, a situation may occur where a message traverses the same node twice, first on the way from the source to its cluster head, then on the way from the cluster head to the appropriate gateway node of the cluster. A similar situation may occur when a message is delivered from one gateway node towards a cluster head and then from the cluster head to another gateway node. If these two gateway nodes are in a common subtree of the cluster, the nodes connecting the common subtree with the cluster head are traversed twice.

A solution to this problem could be to introduce route update messages that are generated at a node that is traversed twice and that are used to shorten the path by the unnecessary nodes. For example, such a route update message could be sent to the node of this cluster that received the message first. This node could use this extra information to deviate from the standard behaviour of forwarding all messages to the cluster head. It could directly address the message towards the outgoing gateway of the cluster, leaving out the nodes that have been passed twice before. Such a route update is only useful for consecutive messages between the same source and destination nodes, as the first message will have to use the strictly hierarchical route in order to discover that some nodes are used twice.

Another solution to this problem would be to use the information gathered by ReBaC2 more efficiently. When a node is a gateway, all of its direct and indirect parent nodes gain knowledge about its gateway status, because this information is aggregated and sent towards the cluster head. This information is currently only used for informing the cluster head about gateways in the cluster. If this information is also used for routing decisions in the nodes, a data message does not have to be sent towards the cluster head if the destination of the message lies within the subtree of the current node. A node can directly forward the message into the subtree that contains the destination node or gateway, and the detour via the cluster head is not necessary anymore.

Bibliography

[Bas99] S. Basagni. Distributed clustering for ad hoc networks. In *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 310–315, Perth, Western Australia, June 1999.

[BFE84] D. J. Baker, J. A. Flynn, and A. Ephremides. The design and simulation of a mobile radio network with distributed control. *IEEE Journal on Selected Areas in Communications*, 2:226–237, January 1984.

[BR02] Elizabeth M. Belding-Royer. Hierarchical routing in ad hoc mobile networks. *Wireless Communications and Mobile Computing 2002*, 2:515–532, 2002.

[FGG07] Imgmar Fliege, Alexander Geraldy, and Reinhard Gotzhein. Micro protocol based design of routing protocols for ad-hoc networks, 2007.

[FGW06] Ingmar Fliege, Rüdiger Grammes, and Christian Weber. ConTraST – a configurable SDL transpiler and runtime environment. In R. Gotzhein and R. Reed, editors, *System Analysis and Modeling: Language Profiles*, volume 4320 of *Lecture Notes in Computer Science*, pages 216–228. Springer Berlin / Heidelberg, 2006.

[Haa97] Zygmunt J. Haas. A new routing protocol for the reconfigurable wireless networks. In *Proceedings of the 6th IEEE International Conference on Universal Personal Communications (ICUPC '97)*, volume 2, pages 562–566, San Diego, CA, USA, October 1997.

[Hei07] Christoph Heidinger. Clustering in mobile ad hoc networks. Projektarbeit, Technische Universität Kaiserslautern, Oct 2007.

[HT98] Ting-Chao Hou and Tzu-Jane Tsai. Adaptive clustering in a hierarchical ad hoc network. In *Proceedings of International Computer Symposium*, pages 171–176, Dec. 1998.

[HT01] Ting-Chao Hou and Tzu-Jane Tsai. An access-based clustering protocol for multihop wireless ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 19(7):1201–1210, July 2001.

[JC03] Tomas Johansson and Lenka Carr-Motyčková. On clustering in ad hoc networks. In *Proceedings of the Swedish National Computer Networking Workshop*, August 2003.

[KGGR05] Thomas Kuhn, Alexander Geraldy, Reinhard Gotzhein, and Florian Rothländer. ns+SDL – the network simulator for SDL systems. In A. Prinz, R. Reed, and J. Reed, editors, *SDL 2005: Model Driven Systems Design*, volume 3530 of *Lecture Notes in Computer Science*, pages 103–116. Springer Berlin / Heidelberg, 2005.

[MZ99] A. McDonald and T. Znati. A mobility-based framework for adaptive clustering in wireless ad hoc networks. *Wireless Ad Hoc Networks. IEEE JSAC*, August 1999.

[ns2] The Network Simulator – ns-2, Information Sciences Institute, University of Southern California.
http://nsnam.isi.edu/nsnam/index.php.

[PB94] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[PR99] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pages 90–100, New Orleans, LA, USA, Feb 1999.

[SGLA08] M. Spohn and J. Garcia-Luna-Aceves. Improving the efficiency and reliability of the route discovery process in on-demand routing protocols, 2008.

[SH02] Ahmed Safwat and Hossam Hassanein. Infrastructure-based routing in wireless mobile ad hoc networks. *Computer Communications*, 25(3):210–224, February 2002.

[SM02] J. Sucec and I. Marsic. Clustering overhead for hierarchical routing in mobile wireless networks. *INFOCOM*, pages 202–209, June 2002.

[SM04] John Sucec and Ivan Marsic. Hierarchical routing overhead in mobile ad hoc networks, 2004.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall International, Inc., third edition, 1996.

[tau] Telelogic Tau 4.6.4, Telelogic. http://www.telelogic.com/products/tau/index.cfm.

[WNST01] Shih-Lin Wu, Sze-Yao Ni, Jang-Ping Sheu, and Yu-Chee Tseng. Route maintenance in a wireless mobile ad hoc network. *Telecommunication Systems*, 18(1-3):61–84, 2001.