OMOLSR Developer's Guide 1.0

Grup de recerca en Arquitectura i Serveis Telemàtics (AST)
Departament d'Enginyeria Informàtica i Matemàtiques
Universitat Rovira i Virgili
May 2008

Index

1 General Architecture

This middleware is based in offering a channel to enable multi-hop communication in Mobile Ad Hoc Networks. As depicted in Fig. 1, the multi-hop channel (M-Channel) uses both unicast and multicast functionalities provided by the two underlying protocols: jOLSR and OMOLSR.

In first place, OMOLSR computes multicast routing thanks to the information received via events coming from jOLSR. These events allow OMOLSR to update the OMOLSR Network Graph (ONG), which will be used for routing computation as well as for providing membership information. Since OMOLSR is an application level multicast protocol, it will split a multicast message into multiple unicast messages. In order to do this, messages are sent to jOLSR so they can be routed to the other members of the group. Furthermore, message delivery is provided with unicast reliability, so nearly full delivery ratio is achieved.

Besides, jOLSR stores network information in different tables similarly to OLSR specification: Neighborhood information base (NIB) stores neighbor information; Local Link Information Base (LLIB) keeps updated information about the state of links to the neighbors; Topology Information Base (TIB) maintains information of the network topology to perform routing calculation. Apart from these tables, OMOLSR needs a new table called Multicast Group Table (MGT) which stores information about which nodes are present in each multicast group.

Finally, on the bottom layers, UDP Protocol allows jOLSR to send control (broadcast) and data (unicast) packets.

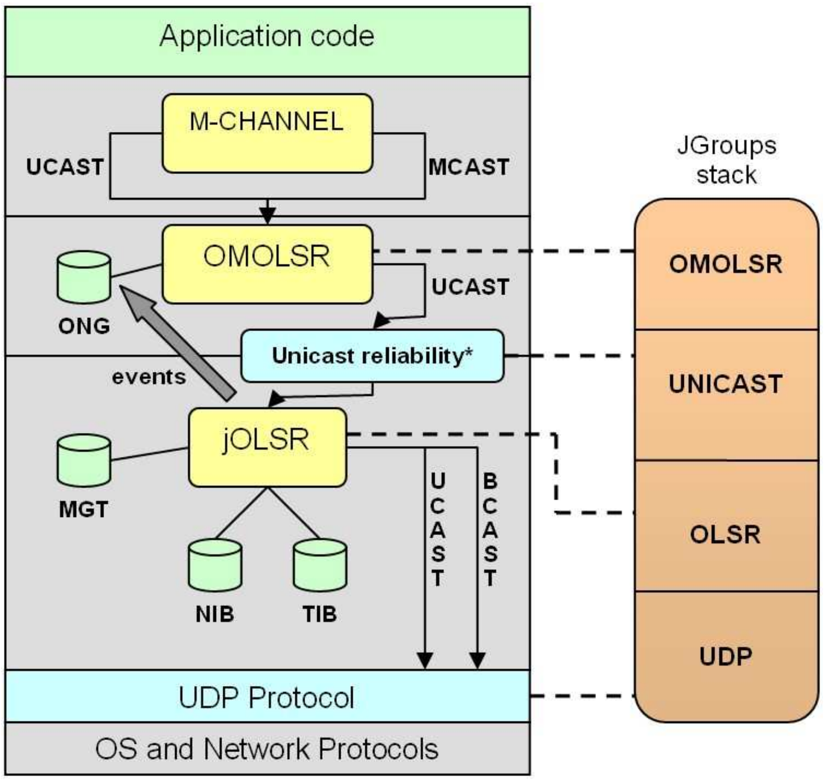

Fig. 1 General Architecture. Each component of the architecture is linked with the corresponding protocol in the JGroups stack

1.1 JGroups

In order to ease and clarify the development of both jOLSR and OMOLSR, we have extended an existing toolkit for reliable communication: JGroups. The key feature of JGroups is its flexible protocol stack, which can be configured and extended depending on the communication needs. Each protocol in the stack provides different functionalities: ordering, reliability, membership, state transfer, etc. In our case, we have implemented both routing protocols (jOLSR and OMOLSR) as JGroups protocols so we can benefit from unicast reliability and ordering by adding the UNICAST protocol to our stack.
Each JGroups channel (JChannel) is initialized with a protocol stack specified in a String or an XML file. The stack is a sequence of protocol names and their optional initialization parameters. For instance, the following stack is used to initialize JChannels in OMOLSR (see urv.conf.ApplicationConfig):

```
“UDP(mcast_send_buf_size=640000;discard_incompatible_packets=true;ucas
t_recv_buf_size=20000000;loopback=false;mcast_recv_buf_size=25000000;m
ax_bundle_size=64000;max_bundle_timeout=30;use_incoming_packet_handler
=true;ucast_send_buf_size=640000;tos=16;port_range=1000;enable_bundlin
g=false;ip_ttl=32;bind_port=5034):OLSR(mcast_addr=224.0.0.10):UNICAST(
timeout=1200,1800,2400,5000,8000;use_gms=false):OMOLSR(mcast_addr=224.
0.0.10)”
```

In OMOLSR, each MChannel is mapped to a JGroups channel.

1.1.1 JGroups dependencies

We have modified some classes of JGroups version 2.5.2 to fit our needs. The source code of JGroups is located under the srcJG252 folder. Modified blocks are noted with a comment in the code (//OMOLSR CHANGE). Anyway, the classes with modified blocks are the following:

- org.jgroups.protocols.UDP: The local address is not obtained from the socket as JGroups does, but is obtained from the protocol stack properties bind_addr and bind_port.
- org.jgroups.protocols.UNICAST: A restriction that does not allow sending messages to non group members is removed.
- org.jgroups.JChannel: User defined events are allowed to be passed up to the MChannel.

2 jOLSR

jOLSR is an application level implementation of the OLSR routing protocol written in JAVA. Although jOLSR includes the basic functionality of OLSR, some modifications have been added to provide topology and group membership information to the upper multicast protocol. For the complete technical specification of OLSR, see RFC3626.
The implementation of jOLSR can be found under the package urv.olsr.

2.1 Data Structures

Data structures used by jOLSR can be found under the package urv.olsr.data. The main data structures are the following:

- NeighborTable: This table contains information about neighbors of the current node and status with all neighbors.
- MprSet: A set that holds information about the multi-point relay nodes (MPRs) of the current node.
- MprSelectorSet: A set that holds information about the nodes that have the current node as MPR.
- NeighborsOfNeighborsSet: This table contains a list of neighbors for each neighbor of the current node.
- TopologyInformationBaseTable: this table stores the information received in TC messages, i.e.: a list of neighbors of each TC message originator.
- RoutingTable: This table defines the routing policy of the node. It defines which is the next intermediate node A that should receive the message when we want to deliver a message to node B.
- MulticastNetworkGraph: This data structure maintains a graph of the network using the information stored in NeighborTable and TopologyTable.
- MulticastGroupsTable: This table contains the list of multicast groups each node has joined.

There are also other auxiliary data structures such as the DuplicateTable, which is used to detect duplicated control messages, or the packet format structures that encapsulate the messages, that will be deeper explained in following sections.

2.1.1 NeighborTable

The NeighborTable is defined in the class urv.olsr.data.neighbour.NeighborTable. Since it is a subclass of ExpiringEntryTable, its entries are removed when the timeout expires. For each node (neighbor of the current node) in the table we store the link status and the neighbors of that node (neighbors of neighbors of the current node).
Whenever the NeighborTable changes (by adding or removing a neighbor or a neighbor of neighbor), a status flag is set to true. This flag is used by other tables that depend on the information stored in this table. After a change takes place, the MPR set must be recomputed as well, and a flag for this action is also set to true.
HELLO messages are created using the information stored in the NeighborTable, and a convenience method is provided to create these control messages.

2.1.2 MprSet

The MprSet is defined in the class urv.olsr.data.mpr.MprSet. This class simply contains a list of the nodes that are chosen as Multi-point Relay nodes.

2.1.3 MprSelectorSet

The MprSelectorSet is defined in the class urv.olsr.data.mpr.MprSelectorSet. This class contains the list of nodes that have chosen the current node as MPR. It also provides a convenience method to create TC messages. Each of these messages include a sequence number that is auto-incremented every time the MPR selector set is changed.

2.1.4 NeighborsOfNeighborsSet

The NeighborsOfNeighborsSet is defined in the class urv.olsr.data.neighbour. NeighborsOfNeighborsSet. The set contains a list of neighbors for each neighbor of the current node. The NeighborTable includes a reference to this set.

2.1.5 TopologyInformationBaseTable

The TopologyInformationBaseTable (or TopologyTable) is defined in the class urv.olsr.data.topology.TopologyInformationBaseTable. Since it is a subclass of ExpiringEntryTable, its entries are removed when the timeout expires.
Entries of this table are indexed by a pair of nodes. A sequence number is used to replace old entries by newer ones.
Whenever the TopologyTable changes (by adding or removing a pair of linked nodes), a status flag is set to true. This flag is used by other tables that depend on the information stored in this table.

2.1.6 RoutingTable

The RoutingTable is defined in the class urv.olsr.data.routing.RoutingTable. The routing table contains the routing information for every node of the network. Each entry of the routing table contains the address of the next hop to reach a specific destination, and the number of hops to the final destination.

2.1.7 MulticastNetworkGraph

The MulticastNetworkGraph is defined in the class urv.olsr.mcast. MulticastNetworkGraph. This class contains a reference to the MulticastGroupsTable as well. The graph is built using the information available in the local Neighbor Table and Topology Table: so the graph is recomputed when changes are detected in Neighbor Table and Topology Table.
This class also provides a method to obtain a contracted graph that only includes the nodes that belong to a specific group. The edges of the contracted graph are tagged with a weight that is the number of physical hops between the vertices of the edge.

2.1.8 MulticastGroupsTable

The MulticastGroupsTable is defined in the class urv.olsr.mcast.MulticastGroupsTable. This class provides methods to register and unregister a node to a specific multicast group, as well as methods to obtain the list of groups a node is registered to.
Whenever the MulticastGroupsTable changes (by adding or removing a multicast group in a node's list), a status flag is set to true. This flag is used by other tables that depend on the information stored in this table.

2.2 jOLSR messages

In order to maintain the routing information up-to-date, each jOLSR node must send control messages to the other nodes of the network. Two types of control message are used in jOLSR: HELLO messages and TC messages.

2.2.1 HELLO messages

A Hello message is periodically broadcasted to the neighbor nodes. It includes a list of neighbors of the local node with the link status of each neighbor. HELLO messages permit a node to know its one-hop and two-hop neighbors, since each node sends information about its local neighborhood.
Hello message is defined in the class urv.olsr.message.HelloMessage. In order to decrease the size of the packets, the message encoding is the following:

- A byte that indicates the number of different linkcodes.
- For each different linkcode, the message includes a byte for the linkcode.
- After the linkcode, the message indicates the size of the list of neighbors which share the same linkcode.
- And finally, the list of neighbors. Each neighbor is represented by its IP address encoded in four bytes.

2.2.2 TC messages

A Topology Control message is periodically broadcasted to all nodes in the network using the multi-point relays (MPRs). This message includes the list of nodes contained in the local MPR selector set, thus topology information of the local node is disseminated to all nodes in the network.
TC message is defined in the class urv.olsr.message.TcMessage. TC Messages perform the same function than in the OLSR specification, but a modification has been added to disseminate information about the multicast groups to all nodes in the network. Therefore, the multicast addresses of the groups joined by the local node are attached in TC messages. This information about the groups each node has joined, is used later to fill the Multicast groups table. Thus, the encoding of TC messages is the following:

- An Advertised Neighbor Sequence Number (ANSN) is associated with the advertised neighbor set.
- The size of the advertised neighbor set.
- The advertised neighbor set: Each neighbor is represented by its IP address encoded in four bytes.
- The size of the list of joined multicast groups.
- The list of joined multicast groups. Each group is represented by its multicast IP address encoded in four bytes.

2.2.3 Message generation

jOLSR needs to send HELLO and TC messages periodically in order to maintain the routing information up-to-date. The class that is in charge of controlling the intervals between message sending is urv.olsr.core.OLSRThread. This class invokes a method on Generator classes that generate a new HELLO or TC message and send it to the neighbors or to all nodes in the network, respectively.
As stated before, HELLO messages are generated in the NeighborTable, since these messages only include information about the local neighborhood.

On the other hand, TC messages are generated in the MprSelectorSet, since these messages include a list of MPR selectors. The information about joined multicast groups is automatically added to the TC messages.

2.2.4 Message handling

When a new message is received, it is processed by different classes and depending on its contents is discarded, delivered to the upper protocols of the stack, forwarded or handled. The overall process is described here:

Messages are received by the OLSR protocol (org.jgroups.protocols.OLSR), which based on the OLSR headers determines whether it is a user data message or a control message. If it is a control message, the OLSRController (urv.olsr.core.OLSRController) is in charge of checking that the message is not a duplicated and its time-to-live is valid. Then, depending on the type of message, the controller invokes either the HelloMessageHandler or the TCMessageHandler.

The HelloMessageHandler (urv.olsr.handlers.HelloMessageHandler) is in charge of updating the data structures taking into account the information contained in the received HELLO message. Data structures that can be modified by the reception of a HELLO message are the NeighborTable, the NeighborsOfNeighborsTable, the MprSet and the MprSelectorSet.

If the received message is a TC message and the current node is a MPR, this message must be forwarded. After checking if the TC message must be forwarded, the message is handled by the class urv.olsr.handlers.TcMessageHandler. This class updates the topology table by removing entries with a sequence number older than the one included in the received message. If the message includes new entries that do not exist in the topology table, these new entries are added to the table.

TcMessageHandler also updates the Multicast Groups Table with the information about the multicast groups joined by the originator of the message.

2.3 jOLSR functioning

jOLSR must maintain tables up-to-date continuously. So, control messages must be sent periodically as well as table information must be removed when timeouts are over. Therefore, jOLSR needs a timing thread that handles these periodic tasks. The implementation of this thread is defined in the class urv.olsr.core.OLSRThread. Tasks assigned to this thread are the following:

- Handle the timeouts of all data structures that extend ExpiringEntryTable.
- Check the flag that indicates whether MprSet should be recomputed.
- Look for changes in NeighborTable, TopologyTable and MulticastGroupsTable (as indicated by their respective change flag). When there are changes in any of these tables, the multicast network graph must be recomputed. If there are changes in the NeighborTable or in the TopologyTable, the RoutingTable must be also recomputed.
- Control intervals between HELLO and TC messages sending.

The algorithm that computes the MprSet is implemented in the class urv.olsr.core.MprComputationController.

Instances and sequence diagram

As aforementioned, JGroups stack is composed by different protocols, one of them being jOLSR. Application messages are passed down the stack and are intercepted by each protocol, which may add a header in each message. Likewise, messages from the network are passed up the stack and protocols may retrieve information from headers of its own type. One application may start more than one JChannel, the basic structure that gives access to the JGroups protocol stack. Therefore, more than one instance of org.jgroups.protocols.OLSR may be started. However, in order to have a single instance of the protocol, the OLSRController is implemented as a singleton. Since this class creates the data structures and starts the OLSRThread, we ensure that there will be only one instance of the core classes of the protocol.

Each JChannel is mapped to a group, and hence to a multicast address. The org.jgroups.protocols.OLSR object registers itself in the OLSRController using this multicast address. The first OLSR object that registers in the OLSRController will be used for message sending. So, the first created channel is the only one used to send messages to the network. The messages sent through the other channels will be redirected to the first channel at jOLSR level. The first channel of each node will always share the same UDP port. This port is used for all message sent, hence the only channel that receives messages from the network is the first created channel.

Message headers associated to each protocol are used to send and retrieve information about these protocols. OLSR headers provide information about the type of message. There are two possible types: DATA (application data) and CONTROL (Hello messages and TC messages). The multicast address of the group (or channel) is also included in the headers. Control messages are delivered two the core classes of the protocol. On the other hand, data messages addressed to the local node are passed up to the corresponding protocol stack depending on the multicast group indicated in the header.

2.4 Unicast routing

The routing of unicast messages in jOLSR is very simple because routing table is continuously updated. jOLSR routes two kind of messages depending on their origin: unicast messages sent down by upper protocols and unicast messages coming from the lower protocols. The former messages are directly routed but the latter are only routed if their final destination is not the local node.

The method OLSRController.handleOutgoingDataMessage() is in charge of sending the message to the next address obtained from the Routing table. The final destination address is kept in a special OLSR header added to all messages, because receiving nodes must know which the actual destination of a message is.

3 OMOLSR

OMOLSR (Overlay Multicast over OLSR) is a new application level multicast routing protocol, designed to work on top of jOLSR. OMOLSR computes locally minimum spanning trees by benefiting from the topology information gathered by jOLSR. The main characteristic of OMOLSR is that it does not need to send additional control packets to perform multicast delivery. The unicast routing protocol already provides all necessary information.

3.1 Data Structures

Data structures of OMOLSR are minimal because all the topology information is provided by the underlying unicast routing protocol, jOLSR. The class urv.omolsr.data.OMOLSRData gives access to the main OMOLSR data structures:

- omolsrNetworkGraph: This object is a representation of the graph that nodes of the multicast group form.
- mstNetworkGraph: This object is a representation of the minimum spanning tree computed from the omolsrNetworkGraph.
- temporalNodes: An auxiliary table that lists nodes that are temporally out of the group membership.

The class OMOLSRData provides the necessary methods to access these data structures and computing minimum spanning tree.

3.2 OMOLSR functioning

The "entry point" of this protocol is the class org.jgroups.protocols.OMOLSR. This class is a JGroups protocol that is included in the protocol stack of MChannel willing to use multicast communication. Like in jOLSR, one application may start more than one JChannel and therefore, more than one instance of org.jgroups.protocols.OMOLSR may be started. But although jOLSR only has an instance of the core classes of the protocol, in OMOLSR there is an instance of each core class for each multicast group.
OMOLSR receives updates of the network graph via events. Once a new update is received, the minimum spanning tree must be computed again.

In the following sections we will explain the basic operations of OMOLSR: dynamic computation of the Minimum Spanning Tree and routing of multicast packets.

3.2.1 Tree computation

When an update of the network graph is received, OMOLSR computes a minimum spanning tree with the local node as the source of the tree, as depicted in Fig. 2. The tree will then be used for routing multicast packets to all the members of the group.

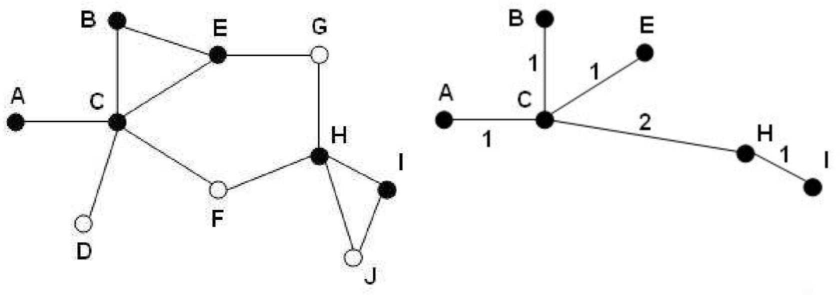

Fig. 2. The graph on the left is the real topology of the network, with the members of the group in black. The graph on the right is the minimum spanning tree for the node A.

3.2.2 Multicast routing

Multicast routing is implemented in the class urv.omolsr.core.StandardHandler. In order to route multicast packets, OMOLSR uses an explicit multi-unicast scheme. When the application generates a new multicast packet, OMOLSR routes the message based on the tree computed for that multicast group. A copy of the packet is sent to each virtual neighbor (a neighbor in the minimum spanning tree), which is responsible for delivering the message to a certain subset of nodes. This subset is defined in each node by using the source-created tree and consists of all the nodes that are in the subtree of each virtual neighbor. This information is then attached to the header of the data packet. When the virtual neighbor receives the message, it computes a tree with the subset of nodes contained in the header. Again, it sends a copy of the data message to its virtual neighbors with new header content. The process is repeated until the subset which must receive the message is empty.

4 Messages and events

The different protocols of a JGroups stack exchange events up and down the stack in order to share information or modify user messages. A message sent by the user's application is itself an event that is passed down through the stack until is sent to the network and is passed up again through the receiver stack.

4.1 Messages

As aforementioned, a message is passed through the stack as a Message Event. At MChannel level we distinguish 2 kinds of outgoing message: multicast message and unicast message. Multicast messages are intercepted by OMOLSR and are split in several unicast packets and passed down. On the other hand, unicast messages are ignored by OMOLSR and are also passed down. So, at jOLSR level, all the packets intercepted are unicast (i.e.: they have a single destination address). Thus, jOLSR simply sends each packet to the next hop to achieve the destination address. It is worth noting that a special protocol header is added to multicast packets intercepted by OMOLSR and unicast packets intercepted by jOLSR.

Received unicast packets at jOLSR level can be unicast packets to forward or, unicast messages arriving at its final destination. However, if these unicast messages have an OMOLSR header, they can also be multicast packets to forward or, multicast messages arriving at their final destination. So, headers are used to mark packets to be intercepted by the protocols.

4.2 Events

OMOLSR uses an event to pass up the network graph from the jOLSR protocol to the OMOLSR protocol and the application layer. This event is encapsulated in the class urv.olsr.mcast.UpdateEvent and is passed up using the USER_DEFINED type of JGroups.

When jOLSR detects changes in the underlying topology or local neighborhood, a new network graph is computed, and a contraction of this graph is passed up to all started channels. Each contracted graph only includes the nodes that are joining the group of the specific channel.

5 Emulator

5.1 Tasks

The emulator accepts task classes that support the emulation and collect statistics.

5.1.1 Topology task

This task, defined in the class urv.emulator.tasks.topology.TopologyChangesTask, is in charge of changing the underlying topology as it is defined in the network graph. This task also checks that the Neighbor table of the jOLSR protocol of each node is consistent with the real neighborhood of each node.

5.1.2 Statistics task

There are two statistics tasks:

- urv.emulator.tasks.stats.CommunicationStatsTask: This task gathers information about all messages sent and received in the network by all applications. It verifies that all nodes that were in the view of the source node received the multicast message.
- urv.emulator.tasks.stats.MembershipStatsTask: This task gathers information about the groups created in the applications and the nodes that joined these groups. This information is checked with the view of each MChannel, in order to verify the correct behavior of getView() method in the channel.

6 Package structure

The implementation classes of jOLSR and OMOLSR are structured under two different top level packages: org.jgroups.protocols and urv.

Under org.jgroups.protocols, we can find the classes that are inserted in the JGroups protocol stack (both protocols: OLSR and OMOLSR). Since these protocols are used in the JGroups protocol stack, they must maintain the package hierarchy specified in the JGroups toolkit. Other classes under this package are the protocol's headers and the SMCAST protocol (a simple multi-unicast protocol that can replace OMOLSR).

Under urv top level package we can find different sub-packages that contain most of the implementation of both protocols. The sub-packages are organized as follows:

- urv.app: This package contains several applications that work on top of the protocols. These applications can be used in emulation and in a real environment.
- urv.conf: This package contains classes that load and store configuration parameters.
- urv.emulator: This package contains the classes that are used in an emulated environment to test applications and protocols.
    - urv.emulator.core: Core classes of the emulator
    - urv.emulator.tasks: This packa
    - urv.emulator.topology:
- urv.olsr: This package contains the main classes of the jOLSR implementation.
    - urv.olsr.core: Core classes of jOLSR: controller, timing thread, table computation.
    - urv.olsr.data: This package holds all data structures used in the protocol.
    - urv.olsr.handlers: this package contains the classes that handle jOLSR control messages.
    - urv.olsr.mcast: This package includes the functionalities added to jOLSR to support upper layer multicast protocols.
    - urv.olsr.message: This package contains the implementation of messages and packet formats used by jOLSR.
    - urv.olsr.util: Utility functions to handle jOLSR data structures.
- urv.omolsr: This package contains the main classes of the OMOLSR implementation.
    - urv.omolsr.core: Core classes of OMOLSR: controller and message handlers.
    - urv.omolsr.data: This package holds all data structures used in the OMOLSR protocol.
    - urv.omolsr.util: Utility functions to handle OMOLSR data structures.
- urv.resources: This package is used to access resources such as GUI images.
- urv.util: This package contains utility classes to operate with dates and graphs.