JOLSRv2 – An OLSRv2 implementation in Java

Ulrich Herberg

October 6, 2008

Abstract

This note describes the architecture of our implementation of OLSRv2 in Java, and some extensions thereto.

1 Introduction

The Optimized Link State Routing Protocol version 2 (OLSRv2) [3] is currently being developed in the MANET working group in the IETF. It offers many advantages over OLSR [4], including support for IPv6, a flexible and extendible message format and fewer message types. This memo describes an implementation of OLSRv2 in Java and several extensions thereto.

1.1 Outline

The remainder of this note is organized as follows. In section 2 the motivation for writing a routing protocol in Java is described, as Java may not be the first language of choice for such low-level software. Section 3 presents the architecture of the implementation that adheres strictly to the separation of the specifications of packetbb [2], NHDP [1] and OLSRv2 [3], and also details each of these three components. In section 4, the general way of extending JOLSRv2 is specified and some of the extensions and Java applets that have been implemented are described. Section 6 presents an extension to JOLSRv2 that allows it to run as routing agent on Ns2, using a library called AgentJ. At last, section 7 describes a network emulator using JOLSRv2 for testing very large arbitrary topologies. Section 8 concludes this note.

2 Motivation

Why would one want to write a routing protocol in Java? There are several reasons that make Java a suitable language for that purpose, at least in a prototype setting. First and most importantly, it is platform independent. Distributing the JOLSRv2 routing protocol is (almost) as simple as downloading a jar file and launching it on the Java Virtual Machine. No complicated `configure` and `make` as known from C/C++ implementations. That also means that code can be handed out to third-parties without imposing the burden of adapting and configuring the source code. The only limitation is that Java does not support manipulation of the routing table, so a small and simple C file using the Java Native Interface (JNI) is provided with JOLSRv2 for adding and removing routes. That file is then all that must be adapted for using JOLSRv2 on a given operating system.

Another advantage of Java is that it is easy to program in and that it offers a lot of existing classes for network operations (e.g. Sockets, DatagramPackets), utility functions such as ArrayLists, HashSets and IO operations (reading and writing files). Therefore, it eases experiments with (prototype) extensions to JOLSRv2, some of which will be presented in section 4.

3 Architecture Overview

According to the separation of OLSRv2 into [2], [1], and [3], JOLSRv2 has been implemented as entirely separated Java projects, related as in figure 1.

The packetbb module is a complete, independent library implementation of [2], with an easy-to-use JavaDoc documented API. This is detailed in section 3.1. The NHDP implementation relies on the existence of the packetbb library and uses

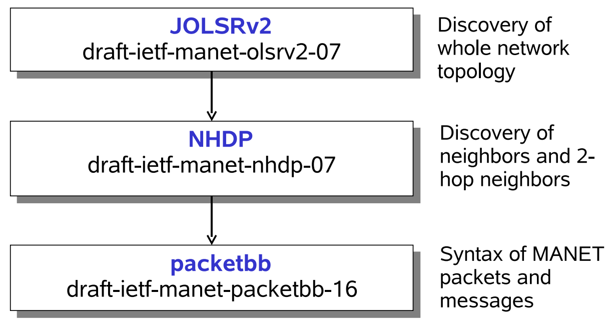
JOLSRv2
draft-ietf-manet-olsrv2-07
Discovery of whole network topology
NHDP
draft-ietf-manet-nhdp-07
Discovery of neighbors and 2-hop neighbors
packetbb
draft-ietf-manet-packetbb-16
Syntax of MANET packets and messages

Figure 1: Architecture of JOLSRv2

the exposed API hereof. It contains all NHDP entities such as node, interfaces, sender and receiver threads, and is detailed in section 3.2. The OLSRv2 module, again, is dependent on the NHDP module and the packetbb library. It uses the existing sets from NHDP, inherits from the NHDP interface and node, and adds all additional functionalities of OLSRv2 (such as sending and processing of TC messages), without rewriting or changing existing NHDP code. A more detailed description of OLSRv2 can be found in section 3.3.

3.1 Packetbb

The packetbb library adheres to [2], with all entities from the specification being realized as distinct Java classes (Message, Packet, TlvBlock, AddressBlock, Tlv). All classes and their class members have been documented using JavaDoc, which allows an easy integration into existing projects. The example in figure 2 illustrates how packetbb can be used.

Note that the information within a Packet object or Message object is not stored in a byte array but in class member variables of the Java class. Thus, before sending a packet, the method packet.getBytes() has to be called which generates the byte array for sending. This enables easy manipulation of a packet or message, at the expense of some extra time required for generating the byte array when sending the message.

The packetbb implementation also allows for parsing only parts of a packet, namely the packet header, or packet header plus message headers. This reduces the time spent on parsing messages that are only forwarded (e.g. incoming TC messages).

```
Message hello = new Message();
hello.setType(HELLO);
hello.setHopLimit(1);

TlvBlock msgTlvBlock =
    new TlvBlock(MESSAGE);

Tlv valid =
    new Tlv(MSG, VALIDITY_TIME);
valid.setValue(TimeTlv.encode(6,C));
msgTlvBlock.add(valid);

hello.setMsgTlvBlock(msgTlvBlock);
byte[] bytes = hello.getBytes();
```

Figure 2: Example: Usage of packetbb

3.2 NHDP

The NHDP implementation mainly consists of the two classes NHDPNode and NHDPInterface, and of classes for all sets specified in [1] as illustrated in figure 3. The classes representing the sets and their tuples are all derived from a class called AbstractSet and AbstractTuple respectively. Again, duplicated code for accessing the sets is avoided by using Java inheritance.

Parameters for the node and each of its interfaces can be set from a config file that is read in at startup. For example, the names of the MANET interfaces (e.g. wlan0) and their IP addresses can be specified. For changing parameters in a running instance of NHDP (and OLSRv2 respectively), a Remote Method Invocation (RMI) stub has been added. This allows for display of all sets, change of any parameter (such as `HELLO_INTERVAL`), addition or removal of interfaces and IP addresses remotely. A client accessing this information is described in section 5.1.

3.3 OLSRv2

The OLSRv2 module is a separate module which uses the NHDP module and the packetbb library. Depicted in figure 3, OLSRv2 has two main classes (OLSRv2Node and OLSRv2Interface) that are each inherited from NHDP. Thus, almost no duplicate code needed to be written such as for sending and receiving messages and for generating or processing HELLO messages. The OLSRv2 module basically consists of all additional

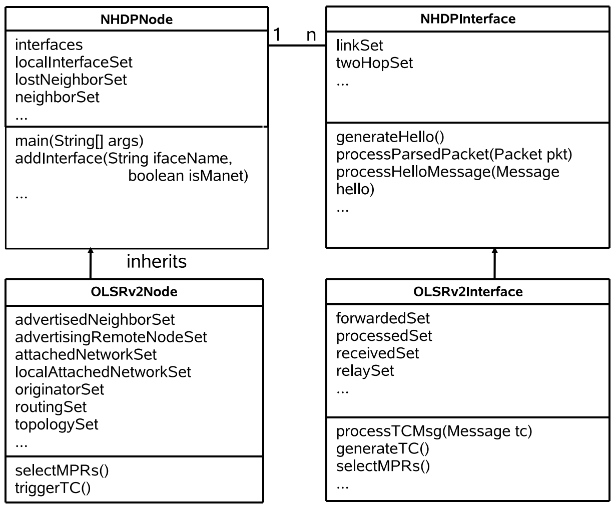

Figure 3: Simplified UML diagram of the most important classes of JOLSRv2

sets that are specified in [3], all new variables and constants (such as `TC_INTERVAL`), and the generation and processing of TC messages.

4 Extensions

Similar to the way that OLSRv2Node inherits from NHDPNode and OLSRv2Interface inherits from NHDPInterface, extensions can be easily integrated into both OLSRv2 and NHDP. Any method that needs to be changed can be overwritten, for example *generateHello()*. As one possible extension, we have implemented a signature mechanism for packetbb messages that allows messages to be signed and verified on sending and reception respectively. This is simply done by inheriting from OLSRv2Interface and overwriting *processHelloMsg()* and *generateHello()*. In order to run this new extension, the node and interface class must be defined in the config file that is read on start. For our signature class for example, the config file would define the parameter `nodeclass = net.jolsrv2.SignedOLSRv2Node` and in the interface section `interfaceclass = net.jolsrv2.SignedOLSRv2Interface` and all necessary parameters such as public and private key. The *main()* function of the Java application then invokes the given node and interface class instead of the default OLSRv2Node and OLSRv2Interface classes. For a detailed description of the signature extension, including simulation results, see [6].

5 Java Applets

Another benefit of using Java, is that it allows for easy web integration by writing Java applets using the packetbb library and/or the NHDP/OLSRv2 modules. Some of these applets are available for the OLSRv2 Interop workshop 08 and will be described in the following sections. All applets can be found on my webpage [5].

5.1 GUI Client

As outlined in section 3.2, our implementation of NHDP and OLSRv2 hosts a Java RMI server. This allows for accessing methods over a stub. Objects that have to be transferred, simply have to implement the *Serializable* interface and are (de-)serialized automatically by Java. The Java applet can then connect over TCP/IP to the RMI server and call all methods that are defined in the stub. Written in Java Swing/AWT, it regularly acquires all sets from OLSRv2/NHDP, and displays these using their *toString()* methods. In addition, all parameters can be changed and new interfaces and IP addresses can be added or removed. We find this to be a useful tool for experimental runs of a given test setup – for a deployment, this feature needs to be removed or authenticated for security reasons.

This applet also allows for displaying the local topology from the point of view of the node to which the applet is connected as depicted in figure 4. It uses a library called Jung2 [9], that enables for drawing graphs and layouting them "optimally" if not geographically correct in space.

5.2 Parser and Packet Creator

The packetbb parser and packetbb creator applet are available on the OLSRv2 Interop08 website[1]. The two Swing/AWT applets use the packetbb module for alllowing easy online interoperability tests of different packetbb implementations. The parser applet has an input field for a string of hexadecimals numbers that represent a (hopefully) valid packetbb packet. Pressing on a button parses the string and creates a Packet object – or displays an error if the packet is not valid. A

[1]4th OLSR Interop / Workshop, Ottawa, CA, 2008, http://interop08.thomasclausen.org

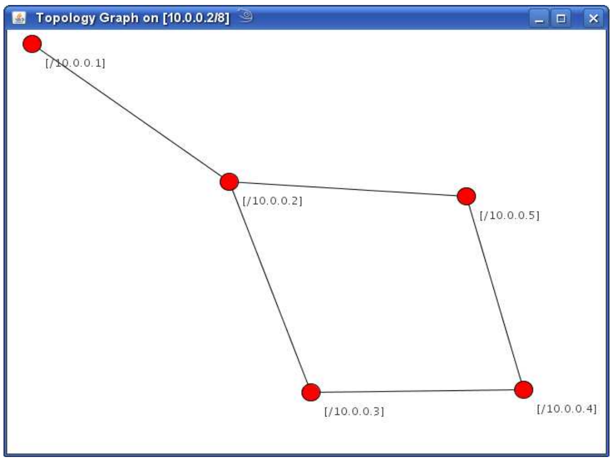

Figure 4: Local view of the topology in the GUI applet

human-readable output of the packet is then displayed. The objective of this applet was to provide a ”head start” on assuring interoperability of OLSRv2 prior to the Interop’08.

The packet creator applet allows for creating packetbb packets by a graphical interface. After creating a packet, the hexadecimal dump as well as a human-readable output is shown.

5.3 Ns2 Trace File Visualizer

This applet, depicted in figure 5, allows for importing Ns2 trace files and to display the node topology over time (similar to ”nam” or ”iNSpect” [7]). Contrary to the latter, no config file has to be written, all parameters – including field size and simulation time – are extracted from the trace file. Again, the applet does not need any installation but runs in a web browser with a JVM plugin. If JOLSRv2 is used as a routing protocol, the applet not only displays the positions of the nodes over time, but also draws edges between nodes that are symmetric neighbors or that are contained in another node’s topology set. For this to work, JOLSRv2 outputs the neighbor set and the topology set in the Ns2 trace file whenever either of these changed. An example for such a line in the trace file would be:

```
L 61.0 2 OLSRV2 N 5;6 T 6;7
```

meaning that OLSRv2 logs (’L’) that at time 61.0 seconds, node 2 has the symmetric neighbors 5 and 6, and the destination node 7 can be reached over node 6 in the topology set.

The applet allows for picking one particular node and displaying its radio range as a grey

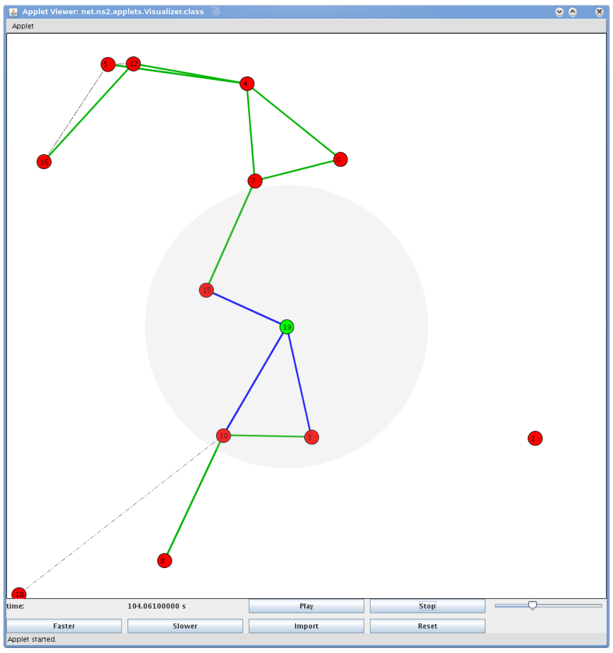

Figure 5: Ns2 trace file visualizer

circle around the node, its symmetric one hop neighbors with blue edges and its topology set with green edges. Thus, the topology of a particular node at a particular time can be easily validated.

6 Ns2 Connection Using AgentJ

One challenge for a routing protocol written in Java is to combine it with Ns2. C or C++ protocol implementations can be used as Ns2 agents in a quite straightforward way by using the Protean library from NRL [8]. This library adds an extra transparent layer between the low level API of the operating system or the Ns2 simulator, respectively, and the higher application levels. Thus, with a single compiler flag change during compilation, the routing protocol can be used either on a real operating system or Ns2.

Running a Java routing protocol in a similar way on top of Ns2 is more challenging. NRL is developing a library for using Java agents in Ns2 called AgentJ [10]. While still under development, this software allows to run Java program without any modification (“as is“) on Ns2. AgentJ uses Java Native Interface (JNI) invocations to launch the Java Virtual Machine and keeps pointers between the Ns2 node in C++ and the Java agent (see figure 6). It uses bytecode rewriting of many java.io and java.net classes for

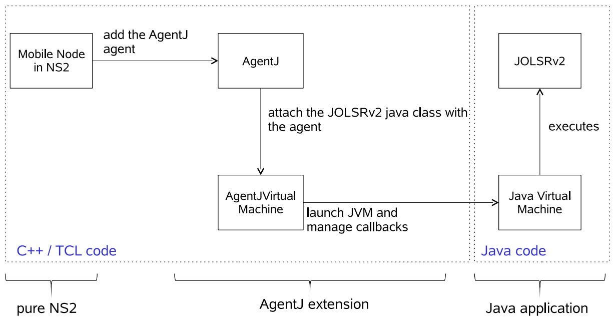

Figure 6: AgentJ integration in Ns2

enabling agents running without change on both real systems as well as Ns2.

However, for running JOLSRv2 on top of AgentJ, modifications were required. First, nodes on Ns2 normally use integer values as node ID. But for testing the effectiveness of packetbb for address compression, our goal was to use IPv6. Thus, when creating the agent, a parameter for the IPv6 address is transmitted and a mapping between this IP address and the node ID is used.

Additionally, a new Java Classloader has been written that allows for using static variables on each node. As all agents are executed in the same JVM, static variables are accessible in the context of all nodes. Values in such a static variable would thus be the same for all nodes. The new Classloader loads all classes from the application and only hands over requests for Sun classes (e.g. java.* and javax.*) to the system Classloader.

At last AgentJ, as is, only allows for Java application agents to run, not for routing agents. So mainly the method *recv()* had to be added for allowing messages to be received and sent. Whenever a node needs to send unicast packets to some destination, *recv()* accesses the Java agent to get the nexthop from the RoutingSet of JOLSRv2. A more detailed description of AgentJ can be found in [10].

7 Emulator

While the Ns2 extension to JOLSRv2, described in section 6, gives the opportunity to study behavior of JOLSRv2 running on mobile nodes, some properties are difficult to test in a discrete event simulator such as Ns2. In particular, Ns2 does not scale up to larger number of nodes, and most simulations will only consider some 100 nodes. For testing an implementation of OLSRv2 concerning efficient in-node algorithms, these small test sets may not be sufficient. In addition, it is difficult (or impossible) to make a statement about timing issues such as the necessary time for RoutingSet calculation, as in Ns2 the discrete time does not advance during this kind of calculation.

For these reasons, an OLSRv2 emulator has been built. It allows to emulate a large topology of nodes. The basic idea is illustrated in figure 7. Note that the emulator runs on **real** machines, i.e. the addresses depicted in the figure have to be bound to a network interface in the operating system. In the figure you can see one node with the address `abcd::1`. This is the node running an OLSRv2 implementation to be tested. On the right side, several nodes are shown with IP addresses from `abcd::2` to `abcd::5` in this example. These nodes shall represent the 1-hop neighbors of `abcd::1`. However, these four nodes are not running on four different machines, but simply on a single machine and a single interface (e.g. eth0) but with four different IP addresses. The emulator running on this machine creates HELLO messages on these four IP addresses and thus emulates four direct neighbors of `abcd::1`. Note that the HELLO messages already include `abcd::1` in the list of neighbors, so that the link is symmetric right from the first HELLO message. In addition, forwarded TC messages of ”virtual“ nodes are created. These ”virtual“ nodes represent nodes that are at least two hops away from `abcd::1`, but are not bound to an IP address on the emulator machine. The four 1-hop neighbors simply create TC messages and pretend that these originate from nodes further away. For example, such a TC message might have a hop count of 4 (i.e. four hops away from `abcd::1`), and any message originator address different from the four direct neighbors.

The emulator can thus pretend to represent arbitrary topologies to the node that is to be tested. An applet has been implemented (using the graph library Jung2 [9]) that allows for creating topologies, exporting these as XML files and reimport these XML files into the emulator to specify the ”emulated topology”. In the applet, the user can simply draw nodes by clicking on an area and creating edges between two nodes by holding the mouse botton. Additionally, random graphs using the Eppstein Power Law or Erdos-Renyi can be created in the applet. As these graphs might be separated into several

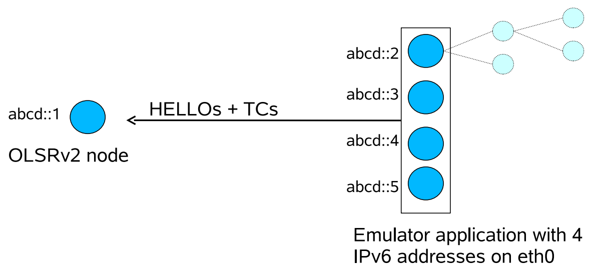

Figure 7: Emulator

connected components, the applet connects these components randomly. Another feature is to import Ns2 tcl files representing a static scenario (i.e. not including any node movements). After having created or imported the graph, a spanning tree with the node `abcd::1` being the root is calculated. This tree is then exported as XML file.

Topologies created with this applet can easily contain several tens of thousands of nodes. This allows for large-scale efficiency studying of OLSRv2 in-node calculations that might not appear in a small test-bed setting and reveal elements (such as route calculation time) that may not be possible using a discrete event simulator.

8 Conclusion

In this paper, an implementation of the routing protocol OLSRv2 in Java has been presented. Java offers a platform-independent, well-documentable way of implementing software and offers many classes for accessing network functionality and other auxiliary tools. The implementation is structured in three separated modules for packetbb, NHDP and OLSRv2. Using inheritance, extensions to OLSRv2 (such as security extensions) can easily be added without rewriting or changing existing code. Another advantage of Java is the ability to write applets that can be called in a web browser. Some such applets presented in this paper allow for online interoperability tests. Using a library called AgentJ, JOLSRv2 can also be used as routing agent in Ns2. This is helpful for researchers interested in using JOLSRv2 for simulating MANETs. Some necessary changes in AgentJ for using it as routing agent (and not only application agent) have been described. At last, an emulator based on JOLSRv2 has been presented that allows for creating huge arbitrary topologies by sending HELLO and TC messages from a single machine but pretending to origin from nodes several hops away.

References

[1] T. Clausen, C. Dearlove, and J. Dean. MANET neighborbood discovery protocol (NHDP). Internet Draft, work in progress, draft-ietf-manet-nhdp-07.txt, July 2008.

[2] T. Clausen, C. Dearlove, J. Dean, and C. Adjih. Generalized MANET packet/message format. Internet Draft, work in progress, draft-ietf-manet-packetbb-16.txt, September 2008.

[3] T. Clausen, C. Dearlove, and P. Jacquet. The optimized link-state routing protocol version 2. Internet Draft, work in progress, draft-ietf-manet-olsrv2-07.txt, July 2008.

[4] T. Clausen and P. Jacquet. Optimized link state routing protocol (OLSR), October 2003. RFC 3626.

[5] U. Herberg. JOLSRv2 applets webpage. http://www.lix.polytechnique.fr/˜herberg/research/jolsrv2/applets/.

[6] U. Herberg and J. Milan. Cryptographical signatures for use in packetbb. Internet Draft, work in progress, draft-herberg-packetbb-signatures-00.txt, November 2008.

[7] S. Kurkowski, T. Camp, N. Mushell, and M. Colagrosso. A visualization and analysis tool for ns-2 wireless simulations: iNSpect. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 503–506, 2005.

[8] Naval Research Laboratory. Protean protocol prototyping library (protolib). http://cs.itd.nrl.navy.mil/work/protolib.

[9] J. O'Madadhain, D. Fisher, and T. Nelson. JUNG – Java Universal Network/Graph Framework. http://jung.sourceforge.net.

[10] I. Taylor, B. Adamson, I. Downard, and J. Macker. AgentJ: Enabling Java Ns-2 simulations for large scale distributed multimedia applications, 2006.