Qolyester: a Modular OLSR Implementation in C++

Ignacy Gawędzki* and Khaldoun Al Agha*†
*LRI Laboratory, University of Paris XI, 91405 Orsay, France
†INRIA Laboratory, Rocquencourt, 78153 Le Chesnay, France

***Abstract*— Thanks to OLSR, proactive routing for mobile ad hoc networks (MANets) is shown feasible and viable. Nevertheless, there are many directions to explore like QoS routing, address autoconfiguration, etc. Therefore, a modular implementation is essential to ease code evolution and feature integration. A modular implementation of OLSR is presented, which is initially aimed at being an experimental platform for OLSR extensions worked on at the LRI. This implementation is written in the C++ language, which allows the use of static genericity paradigms and does not exhibit drastic performance drawbacks.**

I. Introduction

OLSR stands for *Optimized Link-State Routing* and is a proactive routing algorithm for wireless ad hoc networks, which means that the protocol populates the routing tables of each node thanks to the exchange of control messages with other nodes. The link-state type of the protocol implies the diffusion of local topology information in the network to help other nodes reconstructing the global topology on which the routing table entries are calculated.

The strength of OLSR is the way control message diffusion is performed: each node selects a subset of its direct neighborhood as its multi-point relays (MPRs) which are the only neighbors allowed to retransmit control messages. The use of MPRs has been shown to limit the control overhead and make OLSR scalable [1].

To select its MPRs properly, a node acquires knowledge about its 2-hop neighborhood by exchanging HELLO messages with its direct neighbors. Each HELLO message contains a list of direct neighbors heard so far and permits the detection of symmetric links. The MPRs are then selected using a heuristic that aims to select the smallest subset of 1-hop neighbors that together can reach all the 2-hop neighbors.

Another very interesting purpose of the MPRs is that they are the only ones that generate topology messages diffused to the whole network, which further reduces the amount of control overhead.

OLSR does not perform data packet forwarding directly, but instead manages the routing table of the node so forwarding is performed as usual by the IP stack.

Initial motivations are presented in Section II. The general architecture of the project is described in Section III and additional features in Section IV. We then present some aspects related to free software packaging in Section V and finally conclude in Section VI.

II. Motivations

The need for a working implementation of the OLSR protocol was triggered by the SAFARI project[1], which required IPv6 support and the HNA feature, which existing implementations lacked.

In addition, the implementation was to be later augmented with several features worked on at various French laboratories involved in SAFARI (viz node autoconfiguration, multicast routing, QoS routing, security, etc). This suggested a modular approach to software engineering that would make maintenance and evolution easy.

Lastly, because the implementation was aimed at practical applications and not only model validation and simulation, we wanted to have good performances.

All these constraints made us choose C++ as a good implementation choice regarding programming languages.

Widely Used

First of all, almost like its ancestor C, C++ is widely used and compilers benefit from the experience acquired during years of evolution of C compilers which are capable of producing efficient binary code for a large variety of architectures.

Code Security

The typing system of C++ allow programmers to write strict-typed programs if the code respects some fundamental rules. This adds for code security as more errors can be detected at compilation time.

Efficient Containers

The Standard Template Library [2], provides generic building blocks for programming with containers. C++ templates make STL constructs easy to use and yet fully optimizable by the compiler. This allows easy choice of the best internal structure of containers (lists, vectors, trees, hash tables, etc) and easy switch from one to another.

Static Genericity

C++ template evaluation mechanisms turn out to provide a second level of programming language to C++, called **meta-programming** that is evaluated at compile time. The field of meta-programming techniques in C++ is subject to extensive studies [3], [4] and there are a variety of applications of these techniques in situations where performance is the main concern.

[1] `http://www.telecom.gouv.fr/rnrt/projets/res_02_04.htm`

III. General Architecture

A. The Scheduler

Internally, Qolyester is event-driven, which implies the use of a scheduler. Timed events are triggered at some well chosen date, whereas I/O events allow to process data when some input is available.

Timed events are further divided into two kinds: periodic and "once" events. As their names suggest, the former is to be triggered periodically whereas the latter is to be triggered only once. Timed events support the use of a random jitter to be added to the dates of next iteration, as is suggested in the OLSR specification [5].

An I/O event basically needs a set of open file descriptors and flags indicating which operation is to be performed on each file descriptor (read and/or write).

Each event is created with an association to a handler which is to be executed when the event is triggered. Handlers come in various forms, for each type of event and each type of handling that is to be performed. New handlers can be easily added for new purposes.

The scheduling itself is rather simple: file descriptor polling and timed event processing are performed in a loop and core routines are called from inside the event handlers

B. The Core

The core is the more abstract part that performs set and message handling. Reception of messages may change the state of the sets which in turn may imply the transmission of messages.

1) Message Reception: Packets received from the network are handed to the core by the system interface (Figure 1). Packet integrity is then checked and header information is extracted by the packet parser. Each message is extracted and handed to the message parser which checks each message's integrity, extracts header information and hands the payload to each specific message parser (one for each message type).

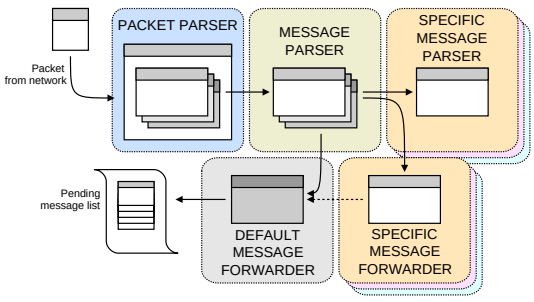

Fig. 1. Message reception

When the specific message parser returns and if the message type indicates to do so, the message is handed to the specific message forwarder. Since the RFC does not specify any other forwarding algorithm apart the default one, each specific forwarder calls the default forwarder.

2) Message Generation: To make information contained in messages as fresh as possible, the payload of locally generated messages is not actually constructed until the time of transmission. A message does not necessarily contain its payload data during its whole lifetime. Data payload is instead generated at the time a packet is built, directly into the packet's data buffer. This not only guarantees that the information is recent, but also avoids unnecessary buffer copying.

In the case of message forwarding, the payload may not be modified by the process and thus a message waiting to be forwarded contains its data payload and buffer copying into packets is unavoidable.

To allow as much messages to go into a single packet and still without delaying message transmission, a queue of pending messages is used (Figure 2). In each iteration of the scheduler loop, the state of the queue is checked and packets are constructed with pending messages and scheduled for transmission. The fact that data of locally generated messages is actually built at the time of packet construction allows partial messages to be generated in the case there is not enough room to put all the information at once.

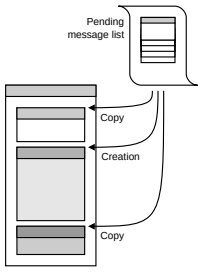

Fig. 2. Message generation

3) Set Handling: According to the RFC, at least the following sets must be maintained:

- Link set
- 1-hop Neighbor set
- 2-hop Neighbor set
- MPR set
- MPR Selector set
- Topology set
- Duplicate set

In addition to that, depending on the optional features included, the following sets are used:

- MID set
- HNA set

Some sets are **correlated** and coherence between entries in them must be ensured. Other sets have elements which should change state or even disappear after a given time. Sets are also used in different ways: all sets support element insertion, but not all of them support explicit element removal (e.g. elements of the Duplicate set are always removed automatically at element expiration). Some sets must support sorted iteration (some must even support different sorting criteria at once) and some must support element search.

The most simple form of correlation is when one set can be seen as a **subset** of another. Then instead of keeping separate instances of the entries inside the set and the subset, the set can be used as the subset as long as a predicate to decide whether an entry is in the subset is provided. To encapsulate this mechanism, Qolyester uses a new design pattern derived from the iterator: the **mask iterator**. If the result of a search is an entry not belonging to the subset, then the mask iterator points to the end of the set to indicate that the element was

not found. If the subset is large compared to the set, in other words if many of the set's elements are also in the subset, then iteration over the subset using the mask iterator is not absurd: when the mask iterator is incremented to point to the next element, it skips all elements not belonging to the subset. A concrete example in Qolyester is the Symmetric 1-hop Neighbor set which is a subset of the 1-hop Neighbor set, the predicate being whether the neighbor is symmetric.

More complicated forms of correlation require the use of a **proxy** object which encapsulates the collection of correlated sets. This ensures that the elements of the correlated sets cannot be modified outside the proxy and the coherence between the sets cannot be broken. Qolyester uses this mechanism for coherence maintenance between the Link set and the 1-hop Neighbor set.

For sets which elements get modified implicitly (either changed or destroyed), a more elaborate solution is used. A reference to elements of such sets are kept in a **Deletable** set along with the date of expiration at which an action has to be performed on the element. A special timed event is triggered at the next date of expiration to call the handler that performs the requested action. Depending on whether the expiration implies some further action on the sets and message generation, a deletable element may request an exact timed action (triggered at the most exact date as possible) or not (i.e. Duplicate set's elements do not require exact removal and can be removed *roughly* after their expiration date is passed).

If a set needs to support sorted iteration, the set is simply implemented as a STL `set` which guarantees that most operations are never worse than logarithmic and that its elements are maintained in sorted order. If a set needs to be iterated on in another sorted order, an **index** is used, i.e. a `set` of iterators to elements of the set. This avoids having to sort the set before iteration and the use of iterators allows to keep only one instance of each element. Of course, a set and all its indexes have to be maintained in parallel to keep coherence and so they are encapsulated in a more general class.

4) Some Details: Some messages' purpose is to advertise the contents of some sets. Since the message payload size can be too small to contain at once all the information to advertise, a form of partial messages has been specified. So only a part of the set's contents can be used to build a message, provided that all the set's elements are advertised once in a given period. To properly advertise all elements equally, a timestamp of last advertisement is used. The message construction then iterates on the set using the index of elements sorted in increasing timestamp order. This provides an elegant way to prioritize elements which have been waiting longer for advertisement. It offers also a way to detect that a additional message have to be constructed immediately otherwise the minimal advertisement period of some elements would not be respected.

C. The System Interface

The interaction with the operating system is rather simple. The operations that the system performs on behalf of the core is essentially packet reception and transmission, low-level network interface information retrieval and setting and routing table management.

All these operations are abstracted in the core and are implemented in a separate part that aims to be as independent of the core as possible. The idea is to enable easy porting to other operating systems (currently Qolyester only runs on Linux) and other improvements (see Section IV).

The core does not use network sockets directly, but an abstract form of network interface instead. An interface is used to send and receive packets and potentially for other network-related operations. This enables, for instance, the use of several sockets instead of only one if so needed[2].

The abstract interface is also used when constructing packets. Indeed, the packet size must not exceed the interface's MTU[3], which has to be retrieved along other interface informations (network address, prefix length, name, etc).

Routing table management is abstracted at the core level as well. Qolyester maintains its own version of the routing tables internally, to allow optimized kernel routing table management. Indeed, only changes of the internal table have to be applied to the kernel table, instead of the table being flushed and populated again at each table change.

IV. Additional Features

A. Address Families

The most obvious feature of Qolyester that is not required by the RFC is that all the core is totally independent of the IP address family (either IPv4 or IPv6). An address is used as an abstract element that supports a range of operations that are common to any family (create, destroy, resolve, convert to string, etc). Nevertheless, to enable an optimal use of the addresses, the choice between IPv4 and IPv6 is static (i.e. made at compile time) to let the compiler inline most of the operations. Since the use of a mixed IPv4-IPv6 ad hoc network makes no sense, there is no need to keep this modularity at run time.

B. Topology Graph

The RFC specifies a way to calculate the routing table based directly on the information contained in the Topology set. While this method gives routes analogous to shortest paths, it restrains the modularity. Qolyester maintains a topology graph along the Topology set and calculates routes on the graph using the Dijkstra algorithm. Moreover, the topology graph is not just another representation of the Topology set since it is also affected by the 2-hop Neighbor set (a route may then exist to a 2-hop Neighbor set even if there are temporarily no MPR to reach it).

Another purpose of the topology graph is the use of QoS metrics on links for QoS routing. Qolyester is aimed at providing an experimental platform for implementing QoS routing techniques, so the path calculation method may not

[2]It has occurred that using multicast in IPv6 for packet broadcast requires the use of one socket for transmission and one for reception to allow fine source address control.

[3]Maximum Transfer Unit: maximum size of a data packet on the network.

necessarily always be a shortest path algorithm. In addition to that, the topology graph could be used to keep all the paths towards destinations in memory, instead of the next hop alone [6].

C. Virtual Interfaces

Since the network interface is abstracted in the core, concrete implementations can be switched in the system interface in order to provide various features without having to modify the core. Pretty early in the development of Qolyester, it appeared that it would be very convenient to be able to run several instances of the daemon on a single computer to make debugging practical. Moreover, some bugs appear only in specific network topologies, which are rather difficult to create using actual separate computers (laptops or PDAs) with WLAN cards. All this led to the idea of **virtual interfaces**.

The general idea is that instead of running the daemons on separate machines and communicating through the radio medium, daemons are run on the same computer and communicate with a **switch** process which purpose is to emulate the network topology. A virtual network interface is then just another implementation of the plain interfaces, supporting all needed operations at higher level, but using communications with the switch instead of actual reception and transmission on the radio interface.

The actual implementation of the switch process is static, i.e. the network topology is provided as a `.dot` file at process startup, but it could be rendered dynamic by implementing some mobility model and making the topology change over time.

If the actual number of Qolyester instances is so large that a single machine could not provide sufficient processing power, the topology of the virtual network could be distributed over several switch instances running on several machines and communicating through a network. The use of a separate switch process is actually the most modular solution.

V. Free Software Package

Qolyester is free software, distributed under the GNU General Public License [7]. Snapshots of the source package are available at `http://qolsr.lri.fr/code`.

A. Autotools

To ease the building of the sources and using of static package options, Qolyester makes use of Autoconf and Automake [8]. The purpose of Automake is to make makefile management as simple as possible, even for large projects with deep source hierarchies. Another advantage of using Automake is that generated makefiles support common targets such as `install`, `clean`, `dist`, etc, to allow easy source code and package management.

Autoconf is used to generate a portable shell script named `configure` that checks for the system's peculiarities and enables the use of workaround code in the sources if possible. The script is called by the user to generate all the files needed for correct compilation (`Makefiles` among others). Another purpose of Autoconf is to let maintainers add global static options easily. The choice of IP address family for instance is made by calling the `configure` script with the `--enable-ipv4` option or not.

B. Doxygen

Doxygen is a tool for documentation generation based on specially formatted comments in the source code. It generates either HTML code or LaTeX source for hypertext and printed manuals. For the time being, only a fraction of Qolyester sources are commented for Doxygen, however the lack of comments does not prevent Doxygen from generating basic documentation which could nevertheless be useful for anyone wanting to dive into the internals.

C. Subversion

Subversion is a Source Code Management (SCM) software [9] aimed at replacing CVS that, in spite of its popularity, lacks a few crucial features. Quite recently, we have put up a public subversion repository at `svn://subversion.lri.fr/qolyester` to allow easy collaboration between partners of the SAFARI project and enable anyone to check out a fresh development version.

VI. Conclusion

Qolyester has now reached a state of maturity that makes it usable in real world applications. It implements all the required and optional features described in the RFC. Anyone is free to download it from `http://qolsr.lri.fr`, compile it and use it.

Future Work

Additional features are to be integrated shortly. Our partners in the SAFARI project are working on address autoconfiguration, multicast routing and security enhancement. We also plan to implement QoS routing using bandwidth and delay metrics as soon as possible. In the QOLSR working group, we are also studying multipath routing as a way to enhance reliability, increase bandwidth and decrease blocking rate.

References

[1] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol," *RFC 3626*, October 2003.

[2] "Standard template library programmer's guide," http://www.sgi.com/tech/stl/.

[3] *Workshop on C++ Template Programming*, 2000, http://www.oonumerics.org/tmpw00/.

[4] *Second Workshop on C++ Template Programming*, 2001, http://www.oonumerics.org/tmpw01/.

[5] T. Clausen and P. Jacquet, "Optimized link state routing (OLSR) protocol," http://www.ietf.org/rfc/rfc3626.txt, RFC 3626, October 2003.

[6] H. Badis, I. Gawędzki, and K. Al Agha, "QoS routing in ad hoc networks using QOLSR with no need of explicit reservation," V. F. '04, Ed. IEEE, 2004.

[7] "The GNU General Public License," http://www.gnu.org/copyleft/gpl.html, Free Software Foundation, June 1991.

[8] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor, *GNU Autoconf, Automake and Libtool*. Pearson Books, October 2000.

[9] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*. O'Reilly Media, 2004.