Development Framework for Supporting Java NS2 Routing Protocols

Ulrich Herberg
Team Hipercom, LIX – UMR CNRS 7161
Ecole Polytechnique
France
Email: ulrich@herberg.name

Ian Taylor
School of Computer Science
Cardiff University
UK
Email: Ian.J.Taylor@cs.cardiff.ac.uk

***Abstract*—This paper presents a framework for developing and executing Java routing protocol implementations within the network simulator NS2. NS2 provides extensive support for developing C++ routing protocols, but has no Java support. In this paper, we describe extensions we have made to the AgentJ toolkit that enable routing protocols to be integrated directly into NS2 without needing to extend the internals of NS2 for each new protocol. The framework defines a reusable AgentJ routing protocol definition that can be used to register new protocols dynamically from within Java code. The actual routing protocol can then leverage the AgentJ toolkit for executing unmodified Java applications in NS2. By means of aspect-oriented bytecode rewriting, AgentJ allows preexisting Java routing protocols, which run on the Internet, to run unmodified within NS2. This powerful system also helps researchers to both understand high-level and algorithmic properties of a given Java routing protocol through the analysis of an NS2 simulation and to rapidly develop and debug new routing protocols through prototyping and experimentation.**

I. Introduction

An important step when designing and developing a new routing protocol is often to verify its behavior in a network simulator, such as NS2 [1]. While network simulators have their limits, especially in terms of the fidelity of the lower layers and, for wireless network interfaces, the fidelity of the propagation model used for representing the behavior of radio waves, their use facilitates the understanding of high-level and algorithmic properties of a given routing protocol. In particular, in the area of Mobile Ad Hoc Networks (MANETs), simulations are often easier to perform than building a large testbed network of nodes, simulating mobility, and guaranteeing the reproducibility of predefined scenarios. Using NS2, scenarios can be defined to simulate not only the network traffic but also the underlying mobility patterns that affect signal strengths between nodes as the scenario progresses. However, NS2 requires that a routing protocol is implemented in C++, which not only restricts a programmer's language choice but also limits the simulations that can be performed for a specific application to be tested; that is, NS2 requires developers to discretize their application into a sequence of events for simulation, rather than running actual code.

Recently, however, there has been a toolkit, developed by the Naval Research Laboratory (NRL), called AgentJ [2], which allows Java applications to be run unaltered within NS2. AgentJ is a set of instrumentation classes that enable unmodified Java applications, which currently run on conventional networks and operating systems such as Linux or Mac OS, to be used within an NS2 simulation. AgentJ uses a combination of Java bytecode rewriting and aspect oriented programming techniques, to convert socket, I/O and timing APIs into low-level C++ methods that bind to the various NS2 counterparts. AgentJ therefore takes an "as-is" Java multithreaded application and converts it into a thread-synchronized "serial" NS2 version, capable of running within a single-threaded discrete time network simulation environment. However, AgentJ originally only targeted the application layer agents, and did not consider the encapsulation of the NS2 routing agents as well. In this paper, we discuss the extensions we made to the AgentJ architecture, which enable it to use Java routing protocols within the NS2 environment and illustrate this routing architecture through the use of specific examples that show how to instantiate and run a new Java routing protocol implemented in NS2.

The paper is organized as follows: After describing related works in section II, and explaining the basic functionalities of AgentJ in section III, section IV details how to use the presented routing protocol extension to AgentJ. Section V describes the architecture of the extension. Section VI analyzes the performance of the extension, and section VII summarizes the features of AgentJ with the extension.

II. Related Work

Network simulations of routing protocols or extensions thereof represent an important part of research in the area of MANETs. According to the survey conducted in [3], 75% of all publications at the MobiHoc conference in the years 2000 to 2005 contain empirical results obtained using network simulations; and further, 44% of which used NS2 simulations. While network simulators have known limitations, they provide a quantitative empirical means of enabling better understanding high-level and algorithmic properties of routing protocols. Due to the importance of analyzing routing protocols in network simulators, it is crucial that the results from the simulation are also adequate for real networks. AgentJ, in this respect, represents a clear novelty, because it allows the simulation of a Java routing protocol implementation, which is

978-1-4244-6949-9/10/$26.00 ©2010 IEEE

the exact same code that will be deployed onto a real network. Therefore, AgentJ allows to measure actual working code, rather than discrete approximations that are converted into a non-threaded discrete network simulation environment in order to meet its specific format.

AgentJ builds on a tool named Protolib [4], which in some respects has similar properties to AgentJ but targets C++ applications instead. Protolib provides an abstraction layer for C++ classes, representing sockets, timers and IP addresses in an abstract and reusable fashion. A C++ application using the Protolib API, can be ported between operating systems e.g. MacOS, Linux and Windows, or to network simulators (OPNET and NS2). However, a routing protocol needs to be re-implemented to use the Protolib APl and cannot use the standard C calls for sockets and timers, etc, and also, although Protolib exposes its API in Java, it does not support unmodified Java code, and any Java application would need to be rewritten to use this API instead.

In the Java world, there are three popular Java-based simulators: JNS [5], JiST [6], and J-Sim [7]. In many ways, these simulators provide similar environments to AgentJ. However, they suffer in two main respects. First, they only support a fraction of the transport protocols and environments that the NS2 framework implements, whereas AgentJ leverages the full stack of NS2 protocols. Second, these are Java-only systems, and thus cannot be extended to support protocols in other languages, such as C++. In AgentJ, a protocol can be implemented in Java or embedded into NS2 in C++ and accessed from within AgentJ. Lastly, neither J-Sim nor JNS allow unmodified distributed Java code to run within their simulator, and although JiST allows unmodified Java code, its current set of supported protocols and environments are limited.

III. AgentJ Overview without Routing Functionalities

AgentJ [2] is an environment that facilitates the running of Java applications as *application agents* within the NS2 [1] simulator. By default, NS2 only supports simulations running either C++ or TCL or a combination of the two. AgentJ builds on this environment to provide a linkage between the NS2 TCL simulation orchestration commands and Java, which then in turn marshals the Java network, timing and System classes into Protolib method invocations. The various levels are shown in figure 1.

The class translating, where possible, is contained within Java e.g. Thread handling, NIO, Monitors, concurrent support, etc. However, network sockets, system time, timers and addresses are accessed through low-level calls using to the underlying C++ code. The implementations facilitates two-way transfers from Java to C++ and C++ to Java, allowing the creation of a Java Virtual Machine from the C++ side and for Java to talk to the underlying C++ objects. A callback system between the C++ (and OTcl) side of NS2 and Java is set up by AgentJ for interaction between NS2 and the Java implementation. Thread handling in AgentJ is managed

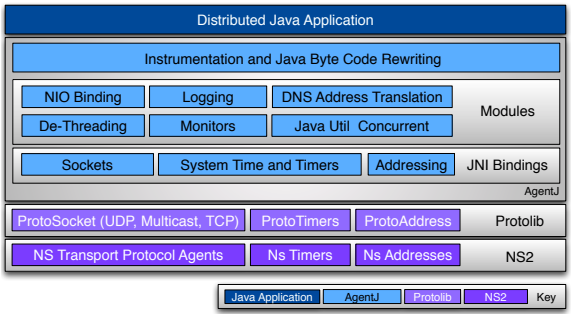

Figure 1. AgentJ architecture outlining its various components

within the rewritten marshaled classes, which performs thread-synchronization on the multiple Java threads. This allows an unmodified Java multi-threaded application, the norm in Java, to be converted into a single-threaded Java applications for simulation. For a detailed description of the rewriting mechanism of AgentJ, refer to [8].

IV. AgentJ Java Routing Protocol Extension

This section illustrates how an existing Java routing protocol implementation can be run on NS2, using the extension that is presented in this paper. Figure 2 depicts the basic routing architecture of a Java application running on a single NS2 node over AgentJ.

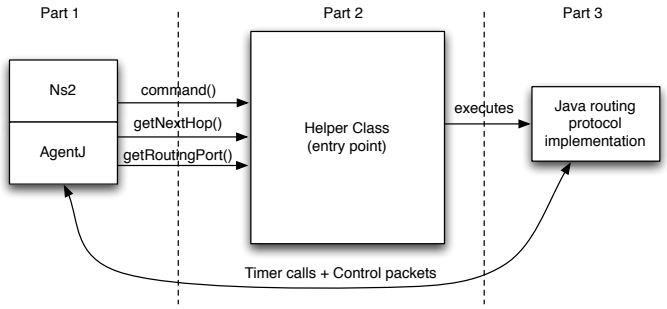

Figure 2. Architecture of an AgentJ agent attached to an NS2 node

Part 1 in the figure shows the NS2 node and the AgentJ agent that is attached to that node, provided by AgentJ. Part 3 depicts the Java routing protocol the user wants to run in NS2. Since AgentJ automatically rewrites the Java bytecode, the Java routing protocol implementation itself does not need to be changed. However, an additional component is needed to provide the "glue" between the Java implementation and the NS2 routing layer, which is depicted in part 2. This linkage is provided by our framework to allow dynamic mapping from multiple Java routing protocols to a single generic AgentJ routing protocol in Ns-2. Further, a message-passing interfaces is provided to allow direct communication with this routing layer, described in more detail in the next section.

A. Implementation of the Helper Class

The helper class serves the same purpose as the `main` method of Java applications: to allow NS2/AgentJ to "execute" the routing protocol implementation. In addition, the helper class

provides an interface with three methods, called by NS2 for the purpose of routing. All other calls, such as for sending and receiving control packets or setting timers, are directly hooked into NS2 and do not need any changes in the Java routing protocol implementation or in the helper class.

The Java helper class (in the following example called `MyRoutingAgent`) should look as the following:

```
public class MyRoutingAgent extends AgentJAgent
implements AgentJRouter
```

The following three methods must be provided by `MyRoutingAgent`:

- `public int getRoutingPort()`
  This method should return the UDP port number that the Java routing agent is running on. This can be any number between 0 and 65535. NS2 will send control traffic packets to that port, and the Java routing protocol should receive control traffic from that port.
- `public int getNextHop(int destination)`
  This method will be called from NS2 whenever a unicast data packet arrives at the node. The Java method should return the next hop for the given destination (as an NS2 node ID) or -1 if no such destination has been found in the routing table. If the Java routing protocol implementation uses real IP Addresses (i.e. "java.net.InetAddress"), this method must perform a mapping between these two address types. An exemplary mapping is:
  `InetAddress.getByName("0.0.0." + destination)`
- `public boolean command(String c, String[] args)`
  The `command()` method evaluates the given command and its arguments. The minimal prerequisite of this function is that it allows to start the routing protocol. In the beforementioned example, calling the command `startRouting` would start the Java routing protocol. Optionally, any other command can be added in the `command()` method, such as for outputting the routing table as in the following example:

```
boolean command(String command, String[] args) {
   if (command.equals("startRouting"))
      // a method needs to be added here
      //     to start the routing protocol
      return true;
   else if (command.equals("print_rtable"))
      // code needs to be added here to
      //   output routing table
      return true;

   return false;  }
```

B. Installation of the Java Classes

In order to run the Java routing protocol, AgentJ has to find the corresponding class files. The location of the Java classes of the routing protocol must be added to an environmental variable called `AGENTJ_CLASSPATH`, e.g.:

```
export AGENTJ_CLASSPATH=.:/path/to/classfiles
```

C. Scenario Tcl Script

The scenario Tcl file that is called by NS2 must tell AgentJ which Java routing protocol to use. Note in particular the following lines, specific to using a routing protocol with AgentJ, that represent a minimum set of parameters that have to be defined in the Tcl script:

1) **Define the routing agent:**

```
set opt(rp)       AgentJ     ;# Routing Protocol
$ns_ node-config  -adhocRouting $opt(rp) \
...
```

The routing agent is set to "AgentJ" whatever Java routing protocol is used. This avoids adding a new routing protocol in `NS2.34/tcl/lib/ns-lib.tcl` for every new Java routing protocol implementation.

2) **Attach the Java agent to a node**

```
set node [ $ns_ node ] ;# create a new node

# The following line must be changed to reflect
# the name of the Java class.
[$node set ragent_] attach-agentj \
                my.personal.MyRoutingAgent

# The following lines need not to be changed
[$node set ragent_] agentj setRouterAgent \
                Agent/AgentJRouter
$ns_ at 0.0 "[$node set ragent_] \
                agentj startRouting"
```

The parameter `my.personal.MyRoutingAgent` must be changed to the name of the Java class name of the helper class that has been added (as described in section IV-A).

D. Change Addressing Scheme

The `$AGENTJ/conf/agentj.properties` file has to be modified as follows:

```
#java.net.preferIPv4Stack=true #or
#java.net.preferIPv6Stack=true
```

In order to use unicast addresses in the routing protocol, the line that corresponds to the preferred address family (IPv4 or IPv6) needs to be uncommented. Note that this modification is to the config files, read at execution time, and so do not require recompilation of neither AgentJ nor NS2 nor the routing protocol implementation.

E. Running the NS2 Simulation

The NS2 simulation can be started by launching:

```
ns my-sample-script.tcl
```

V. Architectural Modifications of AgentJ for Support of Routing Protocols

This section details which architectural changes have been made to in order to support running a Java routing protocol within NS2, depicted in figure 3. On the lefthand side, the NS2 and AgentJ internal objects (`Agentj` and `AgentJRouter`) are depicted. These are implemented in C++ and use some OTcl code.

On the righthand side, the Java helper class that connects the Java routing protocol with AgentJ is displayed. This helper class is a subclass of `AgentJAgent` and implements the Java interface `AgentJRouter`.

In the original AgentJ code, the `Agentj` C++ agent was attached as an application agent to the NS2 node. In contrast, in the modified version of AgentJ with routing functionalities,

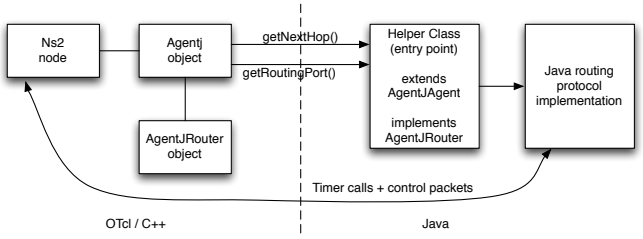

Figure 3. Modified architecture with an AgentJRouter interface

the `Agentj` C++ agent can now also be attached to an NS2 node as a routing agent (i.e. it represents the RTR layer). The `AgentJRouter` C++ class treats packets that arrive at the RTR layer.

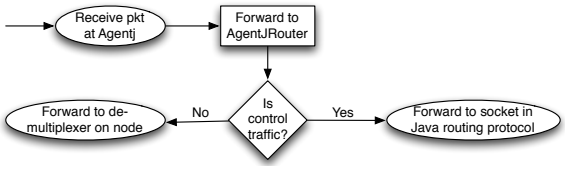

Figure 4. Incoming packet on the RTR layer

Incoming packets are received by the `Agentj` object, and then handed off to the `AgentJRouter` object (as illustrated in figure 4). If the packet is a control packet, it will be handed to a socket on the Java routing protocol. Otherwise, the packet will be treated by the demultiplexer as usual.

Whenever an outgoing packet arrives at the RTR layer (i.e. the `Agentj` object), it is forwarded to the `AgentJRouter` object (as illustrated in figure 5). Then it has to be determined whether the packet is unicast or broadcast. If it is a broadcast packet, the destination is set to the `IP_BROADCAST` address and handed to the LL layer. If the packet is a unicast packet, the `getNextHop()` method on the Java routing protocol is invoked to determine the next hop of the packet. If no next hop if found, the packet is dropped by NS2. This is reflected in the NS2 tracefile as usual (DROP). Otherwise, the next hop is set to the returned next hop, and the packet is handed to the LL layer.

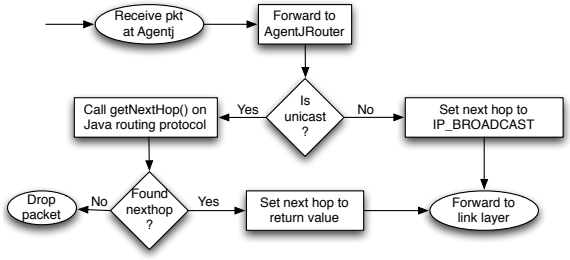

Figure 5. Outgoing packet on the RTR layer

Note that this represents the default behavior of `AgentJRouter` for incoming and outgoing packets, and may be overridden by the user.

VI. Performance Comparison

This section presents a performance comparison between a Java protocol implementation using AgentJ, and a C++ agent. The comparison investigates the time duration and memory consumption of running an NS2 simulation with the respective C++ and Java protocols, performed on a PC with a Core2 CPU 2.1 GHz and 4 GB RAM, with no other time- or memory-consuming processes running.

For this comparison, a basic link state routing protocol has been implemented in both Java and C++. The C++ implementation uses the Protolib [4] API. In order to provide a fair comparison, both versions of the protocol have been implemented as similarly as possible, in terms of structure of the code as well as data structures. The algorithms used are exactly identical (e.g. identical complexity of Dijkstra). It can thus be assumed that both version of the protocol have very similar properties and are well suited for the comparison.

Figure 6 depicts the average time consumption for a single simulation run of the link state routing protocol from 20 to 80 nodes, using the C++ implementation and the Java implementation respectively. Figure 7 shows the memory consumption of both versions, averaged over 20 runs.

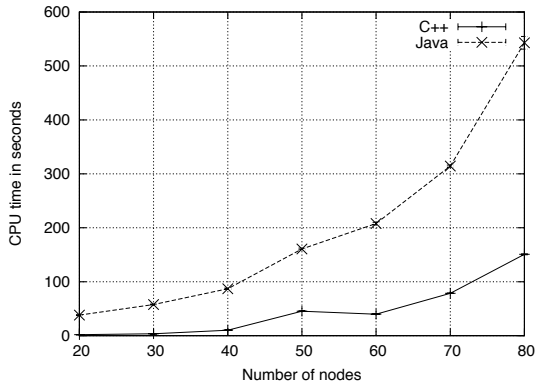

Figure 6. Average CPU time for a single simulation on NS2 with both a C++ and a Java implementation of a link state routing protocol

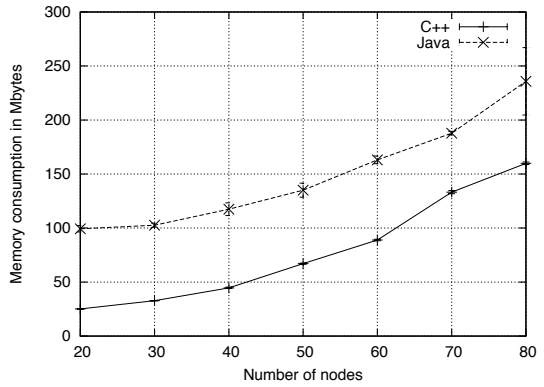

Figure 7. Average physical memory consumption for a simulation run on NS2 with both a C++ and a Java implementation of a link state routing protocol

As can be seen, the Java simulation takes more time and memory than the C++ implementation. This is due to several

factors: First, it is commonly observed that Java is slower than C++ for identical tasks and consumes more memory, so the link state routing protocol itself will be slower. In addition, the thread linearization and bytecode translation for hooking Java commands into NS2 is costly.

It can be observed that the memory consumption difference between the Java and C++ run is almost constant, and the CPU time difference is proportional to the number of nodes. Since commonly simulations are executed by a batch file and can run unattended, e.g. over night, the additional time consumption of Java protocols in AgentJ is acceptable in many cases.

VII. Advantages of AgentJ Routing Framework

The basic version of AgentJ, as described in section III is limited to the application-level agents in NS2 and does not support the execution of Java routing protocols. The modifications presented in section V extend AgentJ's applicability by allowing Java routing protocol implementations to be executed within the same environment in NS2. The main features of this framework are listed in the following:

– Write once, run everywhere: One of the major advantages of Java is that Java bytecode runs on all systems offering a Java Virtual Machine, while the behavior is always the same, which is a very useful property for protocol deployment. AgentJ with the routing extension allows to run a routing protocol implementation intended for a real network additionally in NS2. It is thus possible to verify the correctness, as well as specific properties of the routing protocol implementation in the network simulator without rewriting the code.

– Add routing protocols without modification and recompilation of NS2: For every C++ routing protocol implementation that is added to NS2, parts of the NS2 source code have to be modified, and NS2 has to be recompiled afterwards. With AgentJ and the routing protocol extension, no modification of NS2 and subsequent recompilation is necessary. Multiple Java routing protocos can thus be tested in parallel without modification of NS2, once AgentJ is installed.

– Custom trace format for control messages: Control traffic, that is exchanged between nodes using a Java routing protocol implementation, is traced in the usual NS2 trace files. As such, all evaluation scripts that operate on these trace files, can be used without modifications. If additional tracing of the payload of packets is desired, this can be accomplished as well with a small modification of the trace source code in NS2.

– Custom behavior for treating packets on the RTR layer: For C++ routing protocol implementations, code has to be provided that treats incoming packets on the routing layer. For every such incoming packet, it must be decided whether this packet is a control packet or data packet, whether it needs to be forwarded or sent up to the application layer etc. In AgentJ with the routing extension, a default behavior is already provided, and thus needs not to be supplied by the routing protocol implementation. If a specific packet treatment is desired, however, the packet treating code can be easily extended.

– Unmodified protocol execution: Using other discrete time simulators does not guarantee that the actual algorithm is operating correctly because it does not run an exact copy of the deployed code. Thus, although a simulation may exhibit desirable properties, the implementation of it may not and may contain bugs or suffer from performance inefficiencies. By running the same Java code as in a real deployment with AgentJ, this not only allows accurate simulations to be performed, but it also provides an excellent network debugging environment to analyze any inefficiencies or errors in the code.

VIII. Summary

This paper describes how to run Java routing protocols within the network simulator NS2. The preexisting tool AgentJ allows for using Java agents on an NS2 node, but was not capable of instantiating routing agents. A modification of AgentJ is presented in this paper, which enables the use of Java routing protocols in NS2 without modification of the implemented Java routing protocol, adhering to the Java slogan "write once, run everywhere". In addition, once AgentJ has been installed, NS2 does not need to be recompiled when a new Java routing protocol implementation is added. All parameters that need to be changed can be set in configuration files or in environmental variables, read at execution time. AgentJ, with the proposed routing extension, keeps full compatibility with NS2, meaning that it allows for output of all events in the usual NS2 trace files. Consequently, all evaluation tools used for parsing NS2 trace files can be used without modification.

AgentJ simulations take more time and memory than C++. However, since commonly simulations are executed by a batch file and can run unattended, the additional time consumption of Java protocols in AgentJ is acceptable in many cases.

The routing functionalities have been included in the current distribution of AgentJ that is available for download [8].

IX. Acknowledgements

AgentJ has been development by the Networks and Communication Systems Branch of the IT Division at NRL. Ongoing modifications to the core system is being funded by the Sonoma project, led by J. Macker with associate investigators B. Adamson, J. Dean and I. Taylor.

References

[1] K. Fall and K. Varadhan, "NS-2 web site," http://www.isi.edu/nsnam/ns.
[2] I. Taylor, B. Adamson, I. Downard, and J. Macker, "AgentJ: Enabling Java NS-2 simulations for large scale distributed multimedia applications," in *The 2nd International Conference on Distributed Frameworks for Multimedia Applications, 2006*, pp. 1–7.
[3] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: the incredibles," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 9, no. 4, pp. 50–61, 2005.
[4] "The Protolib Toolkit from the Naval Research Laboratory." [Online]. Available: http://pf.itd.nrl.navy.mil/
[5] "Java Network Simulator." [Online]. Available: http://jns.sourceforge.net/
[6] "JiST: Java in Simulation Time Simulator." [Online]. Available: http://jist.ece.cornell.edu/index.html
[7] "DRCL J-Sim." [Online]. Available: http://www.j-sim.org
[8] Naval Research Lab, "AgentJ: Java network simulations in NS-2. An installation and user manual." [Online]. Available: http://cs.itd.nrl.navy.mil/work/agentj/