Integrating Java Support for Routing Protocols in NS2

Ulrich Herberg

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Integrating Java Support for Routing Protocols in NS2

Ulrich Herberg

N° 7075

October 2009

ISSN 0249-6399 ISRN INRIA/RR--7075--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Integrating Java Support for Routing Protocols in NS2

Ulrich Herberg

Thème : COM – Systèmes communicants
Équipe-Projet Hipercom

Rapport de recherche n° 7075 — October 2009 — 32 pages

Abstract: This document presents a modification of the existing tool AgentJ which allows for running a Java routing protocol within the network simulator NS2.

Key-words: Java, routing protocol, AgentJ, NS2, simulation, tool

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Integrating Java Support for Routing Protocols in NS2

Résumé : Ce document présente une modification de l'outil AgentJ qui permet d'exécuter des protocols de routage en Java dans le simulateur des réseaux NS2.

Mots-clés : Java, protocol de routage, AgentJ, NS2, simulation, tool

1 Introduction

When designing a new routing protocol, an important step in its development is often to implement and experiment with it 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 – in the fidelity of the propagation model used for representing the behavior of physical radio waves, their use is often allowing to understand high-level and algorithmic properties of a given routing protocol. In particular in the area of Mobile Ad-Hoc Networks (MANETs), simulations are easier to perform than building a large test network of nodes, simulate mobility, and guarantee reproducibility of predefined scenarios. Using NS2 as network simulator requires the routing protocol to be implemented in C++. However, there is a tool called AgentJ which allows for running Java applications within NS2.

In particular, AgentJ enables to run unmodified and preexisting Java applications that work on top of an operating system such as Linux or Mac OS also on NS2. AgentJ rewrites all calls related to socket operations, I/O and thread handling, such that the Java application "as-is" interoperates with NS2.

AgentJ was originally developed only for application layer agents, and not for routing agents. In this document, the necessary changes to the AgentJ architecture are described that enable to use Java routing protocols within NS2, as well as detailed instructions on how to instantiate and run in NS2 a new routing protocol implemented in Java.

1.1 Availability

The modifications to AgentJ, that are described in this document, have been integrated into the core distribution of AgentJ. It is therefore available at:

`http://downloads.pf.itd.nrl.navy.mil/agentj/nightly_snapshots/agentj-svnsnap.tgz`

Remark: At the time this document has been written, only the "nightly snapshot" reflects a recent distribution of AgentJ. The "release version" is four years old.

1.2 Notation

Throughout the document, there are two different notations of "AgentJ": AgentJ and `Agentj` (in Courier font). The first notation (with capital "J") is used to describe the whole tool. The latter is used for some of the C++ files that are contained in the AgentJ project.

1.3 Outline

This document is composed of two parts:

Part I details how to instantiate a Java routing protocol in AgentJ. It begins with an overview of AgentJ in section 2. Section 3 contains a step-by-step instruction for integrating a Java routing protocol in NS2 using AgentJ. Section 4 presents some advanced use cases.

Part II details the modified architecture of AgentJ that allows for running routing protocols in NS2. Section 5 overviews routing in NS2. In section 6,

the changes of the AgentJ codebase and architecture are detailed. Section 7 concludes the document.

Note:

- A reader, who only wants to know how to run an existing Java routing protocol implementation in NS2, should read part I.
- A reader, who is also interested in the background and the architecture of routing functionalities in AgentJ, should continue and also read part II.

Part I
Running a Java Routing Protocol in NS2 using AgentJ

This part of the document describes **how** to use a Java routing protocol implementation with NS2 by using AgentJ. It details all necessary steps for integrating the implementation, plus some optional, more advanced features. The background and the architecture of routing functionalities in AgentJ are detailed in part II.

2 AgentJ Overview

AgentJ [5] is a tool primarily developed by Ian Taylor for the Naval Research Laboratory (NRL). The AgentJ tool serves the purpose of running Java applications as application agents on the discrete event simulator NS2 [1]. NS2, without AgentJ, otherwise only supports C++ implementations. AgentJ marshals the Java commands to C++, and rewrites Java application system calls (e.g. for network packets, I/O, and time events) to use the equivalent system calls of NS2. Thread handling is rewritten completely, as NS2 is a single-threaded event-driven simulator, while Java applications often are multithreaded. For a detailed description of the rewriting mechanism of AgentJ, refer to [4]. All this means that an existing Java application does not need to be changed at all for running on NS2, adhering to the Java slogan "Write once, run anywhere".

AgentJ uses the Java Native Interface (JNI) [3] in order to connect the Java code to the C++ code of NS2. That allows NS2 to access of all Java classes, objects and methods. 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.

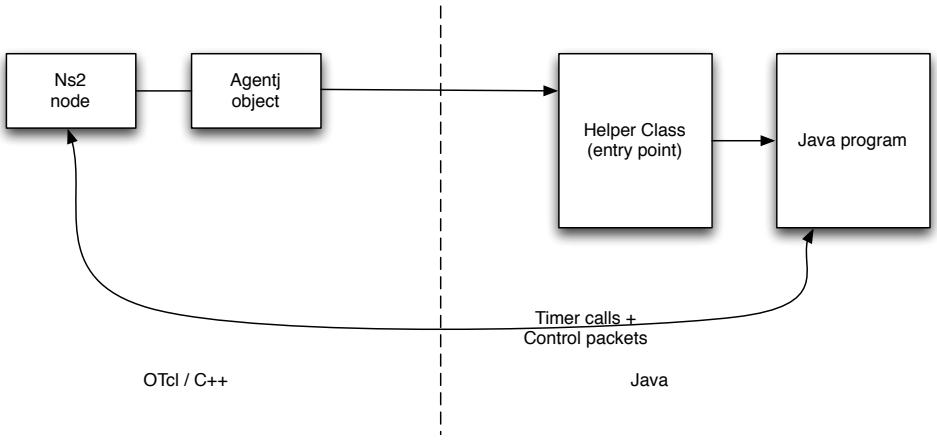


Figure 1: Architecture of an AgentJ agent attached to an NS2 node

Figure 1 depicts the basic architecture of a Java application running on a single NS2 node over AgentJ. The parts on the left side are implemented in C++ and OTcl. This includes the NS2 node and the `AgentJ` C++ agent that is attached to that node. For launching the Java application and sending commands to it from NS2, it needs a single entry point (similar to a `main()` method). Thus, a helper class has to be provided by the user, which serves as this entry point (refer to sections 3 and 6 for more details) and which is the *only* additional component that the user needs to implement to "glue" the Java implementation to NS2.

3 Basic Integration

This section describes the integration of a routing protocol written in Java into NS2 using the AgentJ library as extended as described in this document. It assumes that AgentJ is properly installed, as described in the AgentJ manual [4].

Note that for C++ implementations, each new protocol implementation requires NS2 modifications, and a subsequent recompilation of NS2. However, once AgentJ has been installed, no recompilation of NS2 is necessary any more, even when a new routing protocol implementation in Java is created and added. In the following step-by-step instruction, a demonstrative example routing protocol called `MyRoutingProtocol` is used to show the interaction between the protocol implementation, AgentJ, and NS2.

3.1 Addition of a Helper Class as Single Point of Entry for NS2

As the single point of entry for NS2 to the routing protocol implementation, a new helper class in Java needs to be created. This helper class – as depicted on the right side of figure 2 – 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 two 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 (by virtue of bytecode rewriting) and do not need any changes in the Java routing protocol implementation or in the helper class.

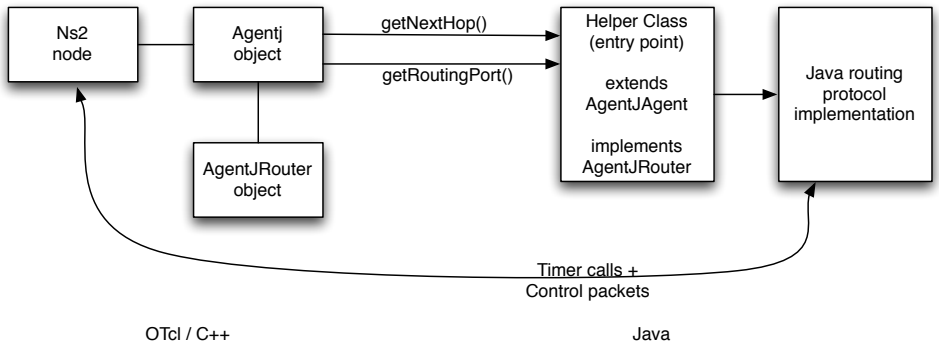


Figure 2: The helper class on the right side serves as single point of entry from NS2 to the Java routing protocol implementation.

The Java helper class (in the following example called `MyRoutingAgent`) needs to extend from `agentj.AgentJAgent`, and also has to implement the `agentj.AgentJRouter` interface. An example class definition looks as the following:

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

By implementing the `AgentJRouter` interface, the three methods `getRoutingPort()`, `getNextHop(int)`, and `command(String, String[])`, must also 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 arbitrary 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 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") and not only NS2 node IDs[1], this method must perform a mapping between these two address types.

  An exemplary mapping (limited to node IDs smaller than 256) would be: `InetAddress.getByName("0.0.0." + destination)`

  In a future version of AgentJ, the mapping could be done within the AgentJ code.

- `public boolean command(String command, String[] args)`

  The method `command(...)` evaluates the given command and its arguments. The minimal prerequisite of this function is that it allows to start the routing protocol (i.e. serves as the single point of entry). 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:

```
public 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;
}
```

[1] In NS2, every node has a unique identifier, the NS2 node ID. Per default, this ID is an unsigned integer (including 0).

3.2 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`. In Linux, this is done by running:

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

where `/path/to/classfiles` needs to be replaced by the location of the Java classes. This line should be added to the `$HOME/.bashrc` file (if using the bourne shell) in order to execute it every time the user logs on.

3.3 Scenario Tcl Script

The scenario Tcl file that is called by NS2 must tell AgentJ which Java routing protocol to use. A complete example Tcl script can be found in appendix A. 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 needs only be changed for advanced settings
   # (refer to section 4).
   [$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 complete name of the Java class name of the helper class that has been added (as described in section 3.1).

3.4 Change Addressing Scheme

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

```
#If you want to change to IPv4 or IPv6 addressing,
#you have to add a line:

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

If the Java routing protocol uses only multicast addressing and no unicast addresses, nothing needs to be changed. In particular, in order to run the AgentJ examples in `$AGENTJ/examples/udp`, the file should not be modified. However, 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 these modifications are to the config files, read at execution time, and so do not require recompilation of AgentJ nor NS2 nor the routing protocol implementation.

3.5 Running the NS2 Simulation

The NS2 simulation can be started by launching the Tcl file

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

The most common error at this point is, that a `java.lang.ClassNotFoundException` is thrown. In that case, make sure that the location of the Java classes of the routing protocol have been correctly included in the `AGENTJ_CLASSPATH` environmental variable, as described in section 3.2.

4 Advanced Features

Section 3 described the basic and necessary steps to interface a Java routing protocol in AgentJ. This section presents some more advanced features of using Java routing protocols in AgentJ.

Section 4.1 explains the steps to include more detailed tracing of control packets. Using this modification allows to output details about the payload of a control packet in NS2 trace files.

Section 4.2 details how to change the specific behavior of the routing protocol when a data packet arrives at an NS2 node. The default behavior of looking up unicast addresses for the destination of a data packet can be modified. For example, for a reactive routing protocol, a route discovery phase could be initiated.

4.1 Tracing of Routing Control Packets

Every control packet that is sent from the Java routing agent will appear in the NS2 trace file[2], just as any control packet sent by a native C++ implementation of a routing protocol. Per default, all control traffic packets sent by the Java agent are marked as `AGENTJ` packets. The following NS2 trace file line represent a typical exemplary output (note that the line-breaks in the following trace entry do not occur in the NS2 trace file).

```
s -t 0.004000000 -Hs 39 -Hd -2 -Ni 39 -Nx 1216.36 -Ny 594.63 -Nz 0.00
-Ne -1.0 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 39.9999 -Id -1.50000
-It AGENTJ -Il 25 -If 0 -Ii 0 -Iv 254
```

Essentially, the line means that a packet has been sent (first letter '`s`') from RTR layer (`-Nl RTR`) at time 0.004 seconds after simulations start (`-t 0.004`). The packet originates from the node with the ID 39 (`-Hs 39`), is destined to port 50000 using the broadcast address (`-Id -1.50000`), and is of type AGENTJ (`-It AGENTJ`). For a detailed description of the trace file format, refer to the NS2 manual [2].

Sometimes, it is desirable to display information from the payload of such a packet. This section describes how to add more detailed packet traces. Note that this advanced modification requires a recompilation of NS2.

Step 1:

If desired, the displayed name of a control packet in the trace file can be changed from the default value `AGENTJ` to any other name. The file `NS2.34/common/packet.h` has to be changed as described in the following:

```
name_[PT_DSR]= "DSR";
name_[PT_AODV]= "AODV";

// change "AGENTJ" to the desired name
name_[PT_AGENTJ] = "AGENTJ";
```

Step 2:

[2]Refer to the NS2 manual [2] for more details about tracing.

For adding a detailed trace of the control traffic packet, the file `NS2.34/trace/cmu-trace.cc` has to be modified:

```
void CMUTrace::format(Packet* p, const char *why)
{
        hdr_cmn *ch = HDR_CMN(p);
        int offset = 0;
        ...
                switch(ch->ptype()) {
                case PT_AODV:
                        format_aodv(p, offset);
                        break;
                case PT_TORA:
                        format_tora(p, offset);
                        break;
```

Specifically, a call to a format method has to be added:

```
                case PT_AGENTJ:
                        format_myrouting(p, offset);
                        break;
```

Next, a method `format_myrouting(Packet*, int)` has to be included in `cmu-trace.cc` that formats the output string in the trace file for the control packet. This method will be similar to the other format methods such as `format_aodv` or `format_tora`. For more information about writing a CMU trace format method, refer to the NS2 manual [2].

In principle, it would also be possible to write a callback handler to the Java routing protocol implementation, which returns a string that will be output to the trace file. This is not part of the current version of AgentJ, but could be added at a later point of time.

NS2 has to be recompiled after these modifications (i.e. `configure` and `make`).

The following NS2 trace file line represent a typical exemplary output after the before-mentioned modifications:

```
s -t 0.004000000 -Hs 39 -Hd -2 -Ni 39 -Nx 1216.36 -Ny 594.63 -Nz 0.00
-Ne -1.0 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 39.9999 -Id -1.50000
-It AGENTJ -Il 25 -If 0 -Ii 0 -Iv 254 -P olsrv2 -Pn 1 -Ps 7996 [-Pt
HELLO -Po 1 -Ph 1 -Pms 41054]
```

All parameters in that line that have a prefix "`-P`", are created by the formating function that has been defined by the user.

4.2 Changing the Behavior of the Router when Data Packets arrive

AgentJ provides a certain default behavior whenever a packet arrives at the routing layer. If it is an incoming packet from the MAC layer, AgentJ will check whether it is a control packet or a data packet. Control packets will be forwarded to the listening socket in the Java routing protocol. This socket is the

one that is bound to the port number given by the method `getRoutingPort()` in the helper class. Data packets are delivered to the application layer of the NS2 node. Outgoing packets will be separated into unicast and broadcast traffic. For unicast packets, AgentJ calls `getNextHop()` of the Java routing protocol to get a valid next hop for the given destination. Section 5 and 6.1 give more details on this.

This particular behavior is implemented in the method `receivePacket()` of the C++ file `$AGENTJ/core/src/main/c/agentj/AgentJRouter.cpp`. This class can be subclassed to change the above-mentioned default behavior for incoming and outgoing packets.

The necessary steps to subclass the `AgentJRouter` class are presented in the following. In this example, the subclass will be called `MyRouter`.

Step 1:

The new files `MyRouter.h` and `MyRouter.cpp` have to be created in the directory
`$AGENTJ/core/src/main/c/agentj`. These files should at least contain the following fields and methods (a complete example of these files can be found in appendix B):

- MyRouter.h:

```
#ifndef _MYROUTER
#define _MYROUTER
#include "AgentJRouter.h"

class MyRouter : public AgentJRouter {
    public:
        MyRouter() : AgentJRouter(){}
        ~MyRouter(){}

        void receivePacket(Packet *p,Handler *handle);
        bool ProcessCommands(int argc, const char* const* argv);
};

#endif // _MYROUTER
```

- MyRouter.cpp:

```
#include "MyRouter.h"

static class MyRouterInstantiator : public TclClass {
    public:
        MyRouterInstantiator() :
            TclClass("Agent/AgentJRouter/MyRouter") {}
        TclObject *create(int argc, const char*const* argv) {
            return (new MyRouter());
        }
} my_router;
```

```
void MyRouter::receivePacket(Packet *p,Handler *handle) {
    // treat an incoming packet
}
```

Step 2:
The new class has to be added to the `Makefile.in` of NS2. The rule can be added to the `OBJ_AGENTJ_CPP` makefile variable:

```
OBJ_AGENTJ_CPP = $(AGENTJ_UTILS)/LinkedList.o \
[...]
       $(JAVM)/TimerWrapper.o $(JAVM)/JAVMTimer.o \
       $(AGENTJ_C_SRC)/MyRouter.o
```

NS2 has to be recompiled after these modifications (i.e. `configure` and `make`).

Step 3: In the scenario Tcl file, AgentJ has to be told to use the subclass instead of the default `AgentJRouter`. At the point in the Tcl file where the nodes are created and the Java agent will be attached, the following line must be changed from:

```
[$node_($i) set ragent_] agentj \
        setRouterAgent Agent/AgentJRouter
```

to:

```
[$node_($i) set ragent_] agentj \
        setRouterAgent Agent/AgentJRouter/MyRouter
```

4.3 Limitations of AgentJ with Routing Functionalities

At the time this document is written, AgentJ only supports routing protocols which send control traffic over UDP. TCP protocols (such as BGP) are not currently supported. In addition, it is currently not possible to run a simulation with nodes using IPv4 and IPv6 addresses at the same time. All nodes have to use the same address family.

Part II
Routing Architecture of AgentJ

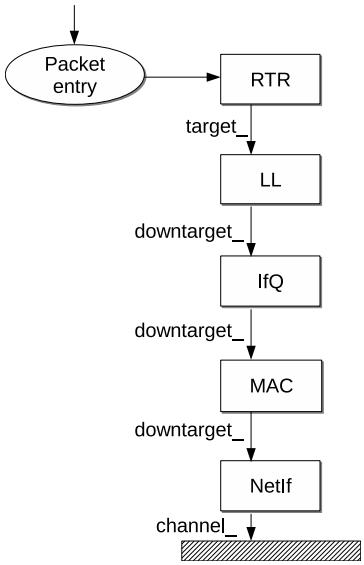
This part presents the background and architecture of routing functionalities in AgentJ. This part does **not** describe **how** to use a Java routing protocol implementation with NS2 by using AgentJ. This was described in part I.

5 Overview of Routing in NS2

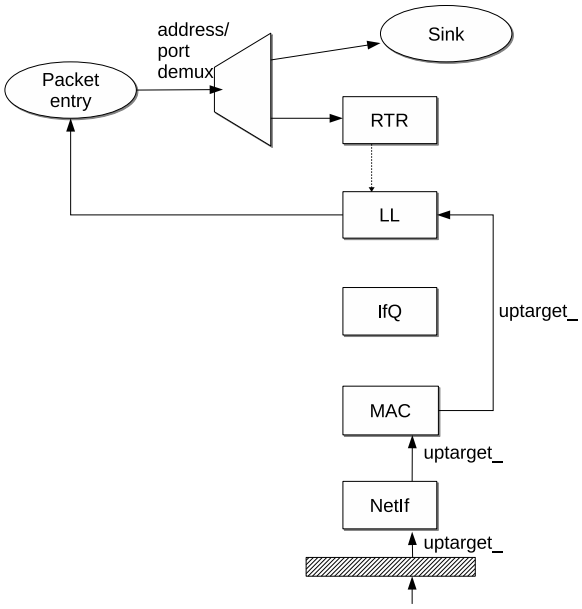
This section presents a brief overview of the routing functionality in NS2. Figure 3 depicts the architecture of routing on an NS2 node and the flow of outgoing and incoming packets. In NS2, each mobile node has attached several layers such as the routing layer (RTR), the link layer (LL), the interface queue (IfQ), the MAC layer (MAC), and a network interface (NetIf).

When a packet is ready to be sent (as illustrated in figure 3(a)), it arrives at what is called Packet Entry in the figure. From there it is handed to the RTR layer which, in turn, has to consider whether it can forward the packet. If the RTR layer finds a valid nexthop for the packet, it hands it down to the LL layer, from where the packet will progress further down the stack towards the channel. Otherwise, it drops the packet. For a more detailed insight into what happens at these layers, refer to the NS2 manual [2].

Figure 3(b) depicts the process for an incoming packet from the channel. The packet is handed to the MAC layer, then to the LL layer, which hands it to the Packet Entry. The demultiplexer has to decide whether this node is the destination of the packet. If yes, then the packet has to be handed to the application agent(s). Otherwise, the packet is to be forwarded along the path to the destination and therefore is to be handed down to the RTR layer. From there, the packet will proceed as depicted in figure 3(a) and described previously.



(a) Outgoing packet



(b) Incoming packet

Figure 3: Layer architecture of a mobile node in NS2 for outgoing and incoming packets

6 Architectural Modifications of AgentJ for Support of Routing Protocols

This section details what architectural changes have been made to in order to support running a Java routing protocol within NS2.

6.1 Overview

Figure 4 depicts the general architecture of the modified version of AgentJ. 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. Section 6.2 details the changes to the C++ objects.

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`. Section 6.3 details the changes to the Java objects.

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, 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 according to the procedure illustrated in figure 5.

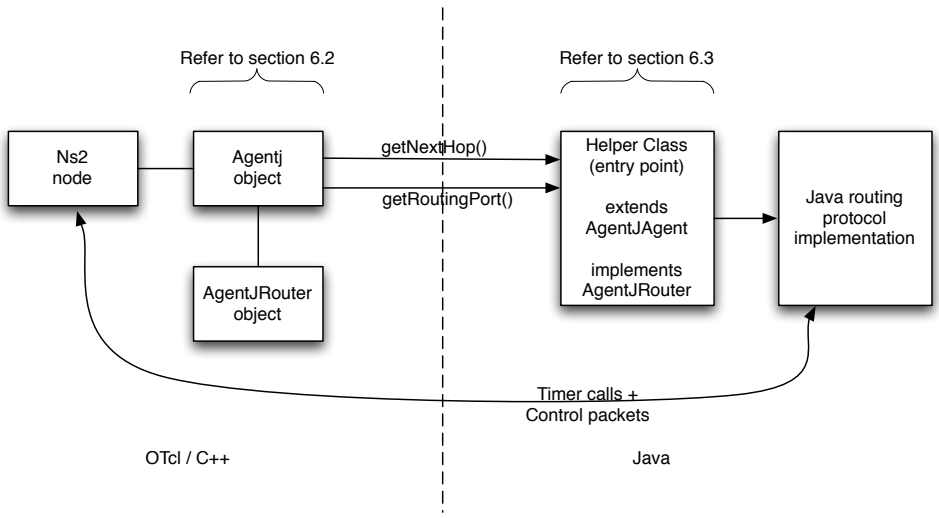
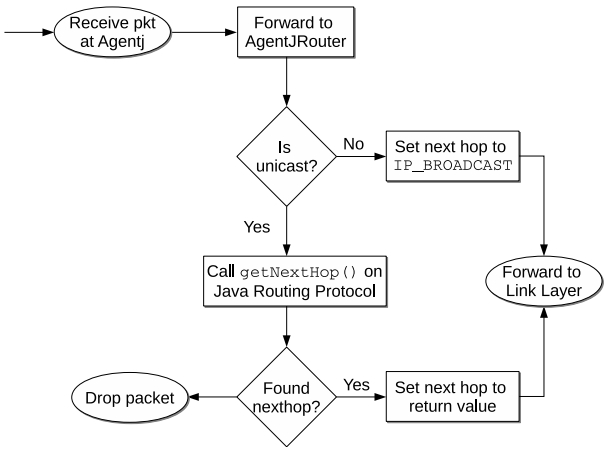


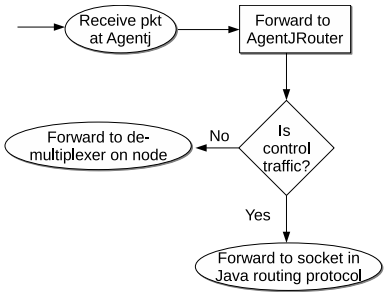
Figure 4: Modified architecture with an AgentJRouter interface

Whenever an outgoing data or control packet arrives at the RTR layer (i.e. the `Agentj` object), it is forwarded to the `AgentJRouter` object (as illustrated in figure 5(a)). Then it has to be determined whether the packet is unicast or broadcast. If the packet 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 address for the destination 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 destination is set to the returned next hop, and the packet is handed to the LL layer.



(a) Outgoing packet



(b) Incoming packet

Figure 5: Decisions to take on the RTR layer

Incoming packets are received by the `Agentj` object, and then handed off to the `AgentJRouter` object (as illustrated in figure 5(b)). 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 described in section 5.

Note that this represents the default behavior of `AgentJRouter` for incoming and outgoing packets, and may be overridden by the user (as described in section 4.2).

In the following subsections, the detailed description of the different new objects from figure 4 is presented.

6.2 C++ Agentj and AgentJRouter Class

In the C++ part of AgentJ, a new class called `AgentJRouter` has been added, and a reference is kept from the `Agentj` class. The reason for extracting the routing handling into a new class, rather than just adding the functionality to the `Agentj` class, is to separate routing tasks from agents that are only used as

application agents. In addition, it allows the user to subclass the `AgentJRouter` in order to modify its behavior.

6.2.1 Modifications of the Agentj C++ Class

The `Agentj` C++ class is the main class of the AgentJ package, which serves as the NS2 agent that can be attached to a node. In order to add the routing functionalities, that class has been changed as described in this section and so to allow AgentJ to be used as both application agent protocol and routing protocol agent. Every C++ object in NS2 that can be accessed by Tcl, provides a method called `command()`. This method presents a general command interface, which allows to send any user-defined commands from NS2 to the routing agent. It receives a string array of commands and executes them. The following Tcl commands have been added in the `command()` function of the `Agentj` C++ class:

- `setNativeAgentRouter`

  This Tcl command creates a new instance of the given C++ class name which must be `AgentJRouter` or a subclass thereof. The command then adds a reference to the `AgentJRouter` object.

- `setNativeRouterPort`

  This call will be forwarded to the `AgentJRouter` object to set the UDP port number it is listening to for control traffic. Incoming packets that have a destination port equal to the port number stored in `AgentJRouter`, will be treated as control traffic and forwarded to the Java routing protocol. Other packets are treated as data packets and forwarded to the demultiplexer of the NS2 node.

- `router`

  All arguments of this command will be forwarded to the `AgentJRouter` and executed as commands on that object.

- `drop-target`

  This sets the drop target of `Agentj` but also of `AgentJRouter`. This is necessary in order to trace dropped packets in the trace file.

- `log-target`

  This sets the log target of `Agentj` but also of `AgentJRouter`. This is necessary in order to trace any events such as sent or received packets in the trace file.

In addition to the changes in the `command()` function of `Agentj`, a new function `recv()` was added. `recv()` represents the Packet Entry as depicted in figure 3. Any packet that arrives on the RTR layer will be delivered to the `recv()` method. As the `Agentj` object is inserted as a routing agent in NS2, but the forwarding decision is taken in the `AgentJRouter` object, this function just forwards the packet to the `AgentJRouter` object.

6.2.2 AgentJRouter Class

`AgentJRouter` is responsible for deciding how to process an incoming control or data packet as described in section 6.1. Specifically, it will acquire the next hop for data packets by calling the appropriate function on the Java routing protocol. The class has the following main methods:

- `receivePacket(Packet *p,Handler *handle);`

  Whenever the `Agentj` class receives an incoming packet at `recv()`, it will call `receivePacket()` on the `AgentJRouter`. This method will treat packets as explained in section 6.1.

- `nsaddr_t getNextHop(nsaddr_t destination);`

  This is called from `receivePacket()` for getting the address of the next hop from the Java routing agent. This method will simply invoke the `getNextHop(int)` method on the Java side and return the result.

- `virtual void forward(Packet *p, nsaddr_t nexthop);`

  After having received the nexthop for a data packet, the packet must be forwarded to that node.

6.3 Changes in the Java AgentJRouter and AgentJAgent

A new Java interface called `AgentJRouter` has been added, and the `AgentJAgent` Java class has been modified to accept some more Tcl commands.

6.3.1 AgentJRouter Interface

This interface is to be implemented by the Java routing protocol and mandates that two methods be provided:

- `int getNextHop(int destination);`

  Called from the native side whenever a (unicast) data packet arrives at the `AgentJRouter` class, the Java routing protocol should return a valid nexthop for the given destination, or -1 if no valid route is found.

- `int getRoutingPort();`

  Returns the UDP port number the routing agent is running on (between 0 and 65535). `AgentJRouter` will decide whether a packet is a data or control packet based on the port returned by this method.

6.3.2 New Tcl Command in AgentJAgent

There is one new Tcl command that the AgentJAgent supports:

- `setRouterAgent`

  This command needs one argument which is the name of the C++ class that will be the `AgentJRouter` (e.g. `Agent/AgentJRouter`).

6.4 Modifications of NS2

A new routing protocol called "AgentJ" has been added to the list of routing protocols in `NS2.34/tcl/lib/ns-lib.tcl` in NS2. The following code, added to `ns-lib.tcl`, will be called whenever a new node is created:

```
Simulator instproc create-agentj-agent { node } {
        set ragent [new Agent/Agentj]
        $self attach-agent $node $ragent
        $self at 0.0 "$ragent startup"
        $node set ragent_ $ragent
        return $ragent
}
```

A new `Agentj` object is instantiated and attached to the node. This agent is told to initialize itself and then attached as a routing agent of the node:

7 Summary

This document 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 document that enables the usage 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.

The routing functionalities have been included in the current distribution of AgentJ that is available for download at:

`http://downloads.pf.itd.nrl.navy.mil/agentj/nightly_snapshots/agentj-svnsnap.tgz`

Acknowledgments

Ian Taylor was very helpful when developing the routing extension to AgentJ. He spent a lot of time and effort on AgentJ, a tool which the author uses intensively for his current research.

References

[1] K. Fall and K. Varadhan. The network simulator – NS-2. http://www.isi.edu/nsnam/ns.

[2] K. Fall and K. Varadhan. The ns manual (formerly ns notes and documentation).

[3] S.Liang. *Java Native Interface: Programmer's Guide and Specification.* Prentice Hall PTR, 1999.

[4] I. Taylor. AgentJ: Java network simulations ins NS-2. An installation and user manual.

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

A Sample Tcl File

```
set opt(chan)   Channel/WirelessChannel   ;# channel type
set opt(prop)   Propagation/TwoRayGround ;# radio-propagation model
set opt(netif)  Phy/WirelessPhy  ;# network interface type
set opt(mac)    Mac/802_11    ;# MAC type
set opt(ifq)    Queue/DropTail/PriQueue    ;# interface queue type
set opt(ll)     LL     ;# link layer type
set opt(ant)    Antenna/OmniAntenna    ;# antenna model

set opt(x)        1000       ;# X dimension of the topography
set opt(y)        1000       ;# Y dimension of the topography

set opt(ifqlen)   50         ;# max packet in ifq
set opt(nn)       5          ;# number of nodes
set opt(lm)       true       ;# log movement
set opt(stop)     30         ;# simulation time
set opt(rp)       AgentJ     ;# Routing Protocol

LL set mindelay_   50us
LL set delay_      25us
LL set bandwidth_  0  ;# not used
Agent/Null set sport_    0
Agent/Null set dport_    0
Queue/DropTail/PriQueue set Prefer_Routing_Protocols    1

# unity gain, omni-directional antennas
# set up the antennas to be centered in the node
# and 1.5 meters above itAntenna/OmniAntenna set X_ 0
Antenna/OmniAntenna set Y_ 0
Antenna/OmniAntenna set Z_ 1.5
Antenna/OmniAntenna set Gt_ 1.0
Antenna/OmniAntenna set Gr_ 1.0

Mac/802_11 set CWMin_               15
Mac/802_11 set CWMax_               1023
Mac/802_11 set ShortRetryLimit_     7
Mac/802_11 set LongRetryLimit_      4
Mac/802_11 set RTSThreshold_        2000
Mac/802_11 set SlotTime_          0.000020        ;# 20us
Mac/802_11 set SIFS_              0.000010        ;# 10us
Mac/802_11 set PreambleLength_    144             ;# 144 bit
Mac/802_11 set PLCPHeaderLength_  48              ;# 48 bits
Mac/802_11 set PLCPDataRate_      1.0e6           ;# 1Mbps
Mac/802_11 set dataRate_          11.0e6          ;# 11Mbps
Mac/802_11 set basicRate_         1.0e6           ;# 1Mbps
Phy/WirelessPhy set CPThresh_ 10.0
Phy/WirelessPhy set CSThresh_ 1.559e-11
Phy/WirelessPhy set RXThresh_ 3.652e-10  ;# = 250 m
Phy/WirelessPhy set Rb_ 2*1e6
```

```
Phy/WirelessPhy set Pt_ 0.2818
Phy/WirelessPhy set freq_ 2.4e+9
Phy/WirelessPhy set L_ 1.0
proc getopt {argc argv} {
  global opt
  lappend optlist cp nn seed stop tr x y

  for {set i 0} {$i < $argc} {incr i} {
    set arg [lindex $argv $i]
    if {[string range $arg 0 0] != "-"} continue

    set name [string range $arg 1 end]
    set opt($name) [lindex $argv [expr $i+1]]
  }
}

proc create-god { nodes } {
  global ns_ god_ tracefd

  set god_ [new God]
  $god_ num_nodes $nodes
}


# ============================================================
getopt $argc $argv

#
# Initialize Global Variables
#
set ns_            [new Simulator]
set chan           [new $opt(chan)]
set prop           [new $opt(prop)]
set topo           [new Topography]

set nf [open MyNamFile-$opt(nn).nam w
set f [open MyTraceFile-$opt(nn).tr w]

$ns_ namtrace-all-wireless $nf $opt(x) $opt(y)
$ns_ trace-all $f
$ns_ use-newtrace

$topo load_flatgrid $opt(x) $opt(y)
$prop topography $topo

#
# Create God
#
create-god $opt(nn)
```

```
#
#  Create the specified number of nodes $opt(nn) and "attach" them
#  the channel.
$ns_ node-config \
                 -adhocRouting $opt(rp) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propInstance $prop \
                 -phyType $opt(netif) \
                 -channel $chan \
                 -topoInstance $topo \
                 -wiredRouting OFF \
                 -agentTrace ON \
                 -routerTrace ON \
                 -macTrace ON \
                 -movementTrace $opt(lm)


# now create nodes
for {set i 0} {$i < $opt(nn)} {incr i} {
   set node_($i) [ $ns_ node ]
   [$node_($i) set ragent_] attach-agentj my.personal.MyRoutingAgent
   [$node_($i) set ragent_] agentj setRouterAgent Agent/AgentJRouter

   $ns_ at 0.0 "[$node_($i) set ragent_] agentj startRouting"

   $ns_ at 0.0 "$node_($i) log-movement"
   $node_($i) random-motion 0
   $ns_ initial_node_pos $node_($i) 20
}

$ns_ at $opt(stop).00000001 "puts \"NS EXITING...\" ; $ns_ halt; "


# Node Initial Positions
$node_(0) set X_ 242.955469762
$node_(0) set Y_ 987.493442159
$node_(0) set Z_ 0.000000000
$node_(1) set X_ 225.147406876
$node_(1) set Y_ 274.091839102
$node_(1) set Z_ 0.000000000
$node_(2) set X_ 170.330849182
$node_(2) set Y_ 525.001140426
$node_(2) set Z_ 0.000000000
$node_(3) set X_ 573.262627815
$node_(3) set Y_ 626.589897443
$node_(3) set Z_ 0.000000000
```

```
$node_(4) set X_ 969.778307544
$node_(4) set Y_ 310.299546663
$node_(4) set Z_ 0.000000000

# Node Movements
$ns_ at 0.0 "$node_(0) setdest 243.402233718 887.494440154 3.371378059"
$ns_ at 29.6 "$node_(0) setdest 202.222490121 796.366899217 3.442128171"
$ns_ at 0.0 "$node_(1) setdest 261.037139025 180.754134361 4.413709080"
$ns_ at 23.4 "$node_(1) setdest 248.041392110 279.906091286 4.413709080"
$ns_ at 0.0 "$node_(2) setdest 251.526166181 466.628882483 7.977160606"
$ns_ at 13.7 "$node_(2) setdest 151.528816489 467.356930946 7.977160606"
$ns_ at 0.0 "$node_(3) setdest 525.639788716 538.657734976 7.366205977"
$ns_ at 13.5 "$node_(3) setdest 427.546365593 558.091734566 4.646736198"
$ns_ at 0.0 "$node_(4) setdest 999.999900000 311.899405137 2.131682882"
$ns_ at 14.1 "$node_(4) setdest 930.361317453 315.585892139 2.131682882"
puts "Starting Simulation..."
flush stdout
$ns_ run
```

B MyRouter Samples

B.1 MyRouter.h

```
#ifndef _MYROUTER
#define _MYROUTER

/*
 *  MyRouter.h
 *  AgentJ
 *
 *  Created by Ulrich Herberg on 27/11/2008.
 *  Copyright (c) 2008. All rights reserved.
 *
 */

#include "AgentJRouter.h"


class MyRouter : public AgentJRouter {
    public:
        MyRouter();
        ~MyRouter();

virtual void receivePacket(Packet *p,Handler *handle);
virtual bool ProcessCommands(int argc, const char* const* argv);
    void notifyRoutingProtocol(bool success, nsaddr_t address);


};  // end class

#endif // _MYROUTER
```

B.2 MyRouter.cpp

```
/*
 *  MyRouter.cpp
 *  AgentJ
 *
 *  Created by Ulrich on 27/10/2008.
 *  Copyright (c) 2008. All rights reserved.
 *
 */

#include "MyRouter.h"

static class MyRouterInstantiator : public TclClass {
public:
MyRouterInstantiator() : TclClass("Agent/AgentJRouter/MyRouter") {}
  TclObject *create(int argc, const char*const* argv) {
```

```
return (new MyRouter());
}
} protean_olsrv2_router;


MyRouter::MyRouter() : AgentJRouter() {
PLOG(PL_INFO, "MyRouter starting up\n");
}

MyRouter::~MyRouter()  {
}

bool MyRouter::ProcessCommands(int argc, const char* const* argv){
  if (!strcmp(argv[0], "reset")){
    // do nothing for the moment
    return true;
  }
  return false;
}


void MyRouter::receivePacket(Packet *p,Handler *handle) {
  PLOG(PL_TRACE, "entering MyRouter::receivePacket\n");
  struct hdr_cmn *ch = HDR_CMN(p);
  struct hdr_ip *ih = HDR_IP(p);
  nsaddr_t saddr = ih->src_.addr_;
  nsaddr_t daddr = ih->daddr();

  PLOG(PL_TRACE,"%f : MyRouter::receivePacket at %d : Packet received from %d  \
     to %d (port %d), forwards %d\n", Scheduler::instance().clock(), agentjref_->addr(),
     saddr, ih->daddr(), ih->dport(), ch->num_forwards());
  if(ih->dport() == routingProtocolPort_)
     ch->ptype_ = PT_AGENTJ; // set the packet type


  if(ih->dport() == routingProtocolPort_ && saddr != agentjref_->addr()
          && (ch->num_forwards() == 1)) { // send on up to Agent
    ih->ttl_ -= 1;
    PLOG(PL_TRACE, "%f : MyRouter::receivePacket is a control packet\n",
        Scheduler::instance().clock());
    Tcl& tcl = Tcl::instance();
    tcl.evalf("%s %s %d", agentjref_->getTclNodeName(), "agent", routingProtocolPort_);
    const char* tclResult = tcl.result();
    Agent* theAgent = (Agent*)TclObject::lookup(tclResult);
    if(theAgent){
       theAgent->recv(p,0);
    } else {
       Packet::free(p);
       PLOG(PL_TRACE, "MyRouter::receivePacket theAgent is NULL!\n");
    }
```

```
    return;
  }


  else { // send packet back out interface
     PLOG(PL_TRACE, "MyRouter is sending packet back out interface\n");
    /* check ttl */
    if (--ih->ttl_ == 0){
      PLOG(PL_TRACE, "MyRouter is dropping packet because of ttl\n");
      drop(p, DROP_RTR_TTL);
      return;
    }
    //check to see if its a udp packet or a broadcast packet
    if(daddr==(nsaddr_t)IP_BROADCAST &&
         ih->dport()!=-1 && ch->num_forwards() == 0){//it's broadcast
      PLOG(PL_TRACE, "%f : Sending broadcast packet\n", Scheduler::instance().clock());
      Scheduler::instance().schedule(target_, p, 0.);
      return;
    }

    else { //it has single destination find and send on way
      if(ch->num_forwards_==0) {
       ih->saddr() = agentjref_->addr();
      }
      PLOG(PL_TRACE, "%f : MyRouter getting nexthop for %d\n",
          Scheduler::instance().clock(), ih->daddr());
      nsaddr_t nextHop=getNextHop(ih->daddr());
      if(nextHop!=INVALID && //make sure we don't send it back up
           (nextHop!=ch->prev_hop_ || ch->num_forwards_==0)){
       PLOG(PL_TRACE, "%f : MyRouter trying to forward it to %d\n",
           Scheduler::instance().clock(), nextHop);
       forward(p,nextHop);
       return;
      }
      if(nextHop==INVALID){
PLOG(PL_TRACE, "%f : MyRouter dropping packet because nextHop is INVALID\n",
            Scheduler::instance().clock());
      } else if(nextHop==ch->prev_hop_) {
        PLOG(PL_TRACE,
         "MyRouter dropping packet because nextHop == last hop which is %d\n",
            ih->daddr());
      } else {
        PLOG(PL_TRACE, "MyRouter dropping packet because ch->num_forwards!= 0\n");
      }

      drop(p,DROP_RTR_NO_ROUTE);

    } //end else of isbroadcast
  }
}
```

Contents

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399