AgentJ
Java Network Simulations in NS-2

An Installation and User Manual

A PROTEAN Research Group Project

Naval Research Laboratory, Code 5522.

Editor, Ian Taylor(Ian.J.Taylor@cs.cardiff.ac.uk)
Contributors: Brian Adamson, Ulrich Herberg and Joe Macker

August 25, 2011

Contents

Chapter 1

Introduction

This chapter provides a background into the motivation behind the development of the AgentJ framework. AgentJ is essentially a Java Virtual Machine (JVM) for the NS2 simulation environment [**?**]. It allows multi-thread Java networked applications without modification to be orchestrated and run within multiple network configurations in the NS-2 discrete time simulator. This enables Java networked applications to be stress-tested under certain networked conditions before deployment and new research ideas to be proved and demonstrated through performance or functional analysis and visualisation. AgentJ contains a number of UDP, multiucast and TCP examples to demonstrate its usage and it also contains more complex scenarios that contain multiple sockets and threads.

AgentJ contains a bytecode rewriting subsystem that swaps user code on-the-fly to use its own implementations of network, threading and timing functions for use within Ns-2. AgentJ therefore contains a complete native implementation for the *java.net* package via a toolkit called Protolib for integration with NS-2 and also overrides several key objects within the core JVM including *java.lang.Thread* and even *java.lang.Object*. Below are a list of the key features of the system. AgentJ:

- Manages sockets, network addresses, threads, thread synchronisation (wait and notify), timers (Thread.sleep+ others)
- Includes a ns-2 DNS server and a re-implementation of the sockets from java.net.*
- Re-implements java.lang.Thread - for overriding t java threads. Since Ns-2 is not multi-threaded, AgentJ needs to monitor all threads created by the application and internally control their synchronicity.
- Incorporates a re-implementation of java.lang.Object for Java monitors.
- Consists of a combination of bytecode rewriting and aspect-oriented techniques to dynamically swap the run-time environment for Java networked applications, by:
  - performing a find/replace on the bytecode on various packages, e.g. java.net, some key classes java.lang.Object and some classes in java.io.
  - using Javassist to replace some method-level invocations for wait() and notify(), for overloading start() and run() for thread synchronisation and for timing operations e.g. java.lang.Thread.sleep().

- Includes a number of real-world including demos of P2PS Simulations, a complex multi-threaded middleware, without modification of the source code.

1.1 Motivation for AgentJ

AgentJ has been developed primarily by Ian Taylor[1], who has been working with the PROTEAN Research Group in NRL for the past six years (with PIs Brian Adamson and Joe Macker, and earlier also involving Rick Jones). Other key contributors to the project include Ian Downard[2], Ian Wang[3] and Andrew Harrison[4]. The PROTEAN Research Group has used AgentJ in a variety of projects working dynamic networking protocol and middleware issues including SRSS (Scalable Robust Self-organizing Sensor), MODAN (Multi-agent systems Operating within Dynamic Ad hoc Networks), and currently SONOMA (Service Oriented Networking Operating in Mobile Ad hoc) Networks.

One main theme for all of these projects has been to investigate and model, using network simulation tools, lightweight network application discovery mechanisms suitable for application in mobile sensor systems, leveraging self-organizing computer communication networks where possible, based on Mobile Ad-hoc Networking (MANET) routing protocols which operate using wireless communication links and have no centralized administration or control. The PROTEAN group have been considering a complexity of middleware approaches, from simple network services e.g. network name/address resolution, IP multicast, ANYCAST, to potentially heavy-weight, highly stateful, complex agent-based architectures. However, the focus is on relatively lightweight (minimally complex) middleware discovery mechanisms and services which can facilitate publish and subscribe relationships among a set of sensor application peers participating in an MANET network. To this end, we developed AgentJ in order to provide a framework for investigating the possible Java middleware solutions. This work led to some initial tests investigating common peer-to-peer (P2P) techniques for dynamically discovering and connecting the mobile sensor nodes.

P2P applications typically create virtual network overlays for connecting users from highly transient devices and computers behind NAT, firewalls, etc, often referred to as peers at the *edges of the Internet* [?]. P2P applications and middleware e.g. Jxta [?], typically use a combination of discovery techniques in order to connect peers in more decentralised nature than conventional Web-based applications. In the first instance we looked at *unstructred P2P approaches* and the P2PS middleware [?] was selected over potentially more complex systems such as Jxta, because we had strong in-house support for the software. The unstructured P2P approach is interesting because it attempts to address similar unreliable connectivity issues to mobile sensor environments. However, in this regard sensor nets can be more extreme, where nodes not only disappear/reappear frequently, but data rates are continuously changing as the sensors move away from the wireless hubs and other factors, such as battery strength, can affect the type of role the sensor can play within the network.

[1]Ian.J.Taylor@cs.cardiff.ac.uk
[2]iandownard@ieee.org
[3]ianwangcardiff@googlemail.com
[4]Andrew.Harrison@cs.cardiff.ac.uk

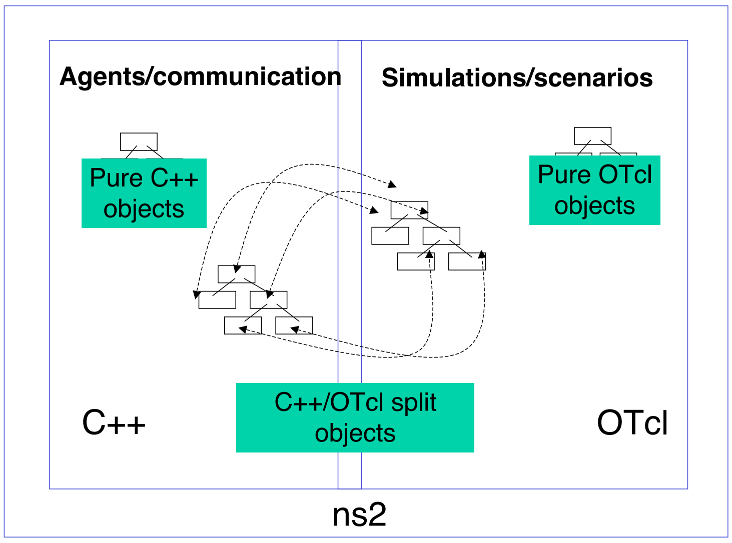


Figure 1.1: Ns-2 Component Overview

1.2 NS-2, Protolib and AgentJ

NS-2 [?] is a discrete event simulator that supports the link layer upwards on the OSI stack i.e. the network, transport, session, presentation and application layer, respectively. It can support both wired and wireless simulations and works on most platforms and therefore satisfies the main focus of the project, that is, to test out various P2P discovery and communication mechanisms within various network extremities. NS-2 builds on years of experience and contains a number of existing transport protocols, written in a combination of C++ and Tcl. The simulation network configuration and orchestration is written in Tcl, the Ns-2 engine is written as a mixture of Tcl (oTcl) and C++ and the underlying protocols are written mostly in C++ with some configuration in Tcl. A rough schematic is provided in Figure 1.1 illustrating these relationships.

The Ns-2 environment was designed for simulating traffic, rather than deal with the content of the traffic. Therefore, in the Protolib toolkit [?], we have extended this vision to implement fairly complete versions of the UDP, Multicast and TCP protocols that permit the sending of payload traffic, which is necessary for message-based systems like P2P and publish/subscribe. For TCP in particular, we have implemented the full range of TCP event handshaking to model real-world TCP behaviour and we have included full support for TCP servers that can handle multiple connections. AgentJ implements its native Java networking binding directly to the Protolib toolkit for interfacing with Ns-2. This is illustrated in Figure 1.2.

Protolib implements a switchable communications layer for several communication protocols (e.g. TCP, UDP, Multicast etc) in that the same programming interface can be used to communicate within NS-2, OPNET and a networked environment. To bind to a particular environment, typically the application just needs to be recompiled. AgentJ re-implements the Java networking, io and threading layers of Java to work with the Protolib toolkit, switched in its NS-2 mode. For networked applications, AgentJ can just switch itself off and Java will default to its conventional native implementations as illustrated.

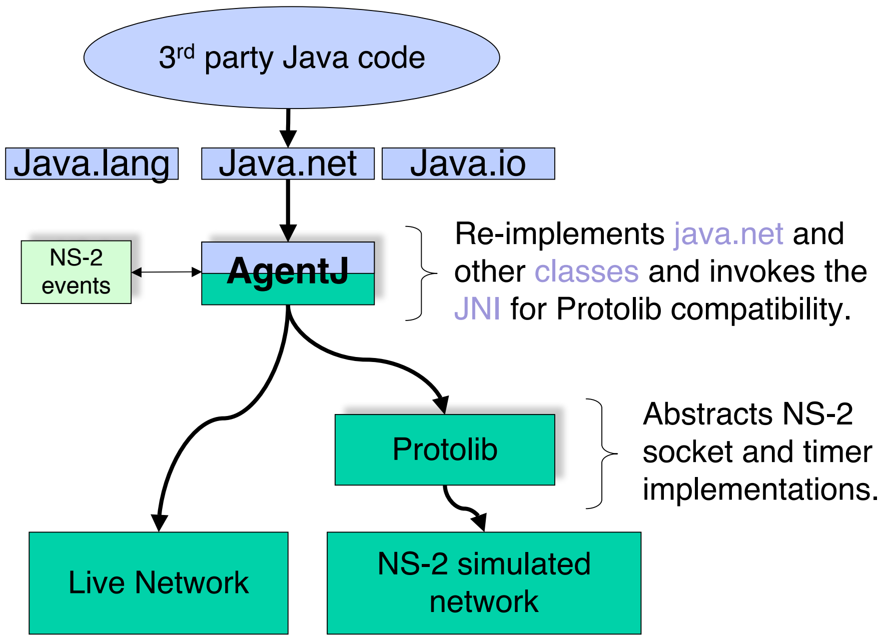

3rd party Java code
Java.lang
Java.net
Java.io
NS-2 events
AgentJ
Re-implements java.net and other classes and invokes the JNI for Protolib compatibility.
Protolib
Abstracts NS-2 socket and timer implementations.
Live Network
NS-2 simulated network

Figure 1.2: An overview of the AgentJ software stack

Since Protolib contains a consistent socket and timer interface to each of its supported environments it would mean that AgentJ could potentially work in OPNET through a recompilation of the underlying shared libraries. Future work may investigate this possibility.

The main thrust of AgentJ lies in the switching between the Java native implementations of its network core and the implementation for NS-2 in Protolib. This is unfortunately non-trivial because there is no such interface in java.net that allows socket implementations to be swapped whilst using non-IP addresses. The current *SocketImpl* and *SocketImpl* mechanism is tied closely into the IP address format, which is not possible to extend to include Ns-2 address scheme (that uses simple integers). Therefore, AgentJ uses bytecode rewriting to provide similar but alternative implementations to create hooks into java.net for access to the different addressing scheme and for implementing a new native implementation for this package.

The result is that Java implementations running in AgentJ are bytecode re-written at run-time in order to invoke the AgentJ native implementation. This means that no prior configuration of alteration of the original source code needs to be undertaken to run these codes in AgentJ. This is a different approach that has been undertaken by other projects (e.g. J-Sim [?] or PeerSim [?]) which involve rewriting a version of an application for simulation.

Chapter 2

Installing the AgentJ Toolkit

This chapter describes the installation of AgentJ toolkit and is divided into two sections. The first describes how to install the native implementation of AgentJ into NS-2, which involves several stages and modification to the Ns-2 makefile, and the second involves the somewhat simpler installation of the Java code.

2.1 Installing the C++ Code

AgentJ has three dependencies, which have to be downloaded and installed **first** before AgentJ can be added. These are:

1. **Ns-2:** Full instructions for downloading and installation of ns-2.34 can be found at the project's Web site at *http://www.isi.edu/nsnam/ns/* or type "ns-2" into Google and hit "I'm Feeling Lucky".

2. **Protolib:** can be downloaded and installed from *http://downloads.pf.itd.nrl.navy.mil/protolib/nightly_snapshots/protolib-svnsnap.tgz* or again "protolib" and "I'm Feeling Lucky" in Google will also do the trick.

3. **JDK6:** is available at *http://java.sun.com/javase/downloads/widget/jdk6.jsp*

The following steps describe how to install Agentj on top of Ns-2:

Step 1:
Download AgentJ: Get a nightly build at

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

Step 2:
First, the Ns2 "Makefile.in" needs to be modified in order to build an AgentJ-enabled version of the Ns2 environment. The *Makefile.in* file can be found in the current release in the ns-2.34 directory of the ns source tree (i.e. ns-allinone-2.34). A copy of my Makefile.in is provided in the nightly build in the agentj/doc/ns2buildfiles/ns234 directory.

Rename the Makefile that is suitable for your operating system (i.e. mac or Linux) to *Makefile.in* and overwrite the ns-2.34 Makefile.in with that file. It is recommended that after downloading and extracting Protolib you put the directory protolib as a subdirectory of ns-2.34 (i.e. ns-allinone-2.34/ns-2.34/protolib). Otherwise you have to modify the *Makefile.in* at the "PROTOLIB Section" section accordingly.

Step 3:

Set the environment variables. You need to set AGENTJ to the home directory of AGENTJ and then add the library path to the library path environment variable (LD_LIBRARY_PATH), and the JAVA_HOME that points to your JDK6, as in the following example for linux. In addition, you have to set your CLASSPATH to include the AgentJ classes. Please change the path names accordingly.

```
export AGENTJ=/home/username/Apps/agentj

export LD_LIBRARY_PATH=$AGENTJ/core/lib/:$JAVA_HOME/jre/lib/i386/: \
     $JAVA_HOME/jre/lib/i386/server:$JAVA_HOME/jre/lib/amd64: \
     $JAVA_HOME/jre/lib/amd64/server

export JAVA_HOME=/usr/java/jdk-1.6.0/

export AGENTJ_CLASSPATH=/path/to/your/java/classes
```

where:

- **AGENTJ:** is used to specify the installation directory of the agentj package. This is used by the Makefile.in NS-2 makefile and also used within the other environment variables defined here.
- **LD_LIBRARY_PATH:** the standard environment variable for specifying where to find libraries. Here, I extend this to include the AgentJ lib directory.
- **JAVA_HOME:** the standard environment variable for specifying the path to your JDK 1.6.
- **AGENTJ_CLASSPATH:** this points to the classes (or jar files) containing your Java applications that you want to run in Ns2.

Step 4: Copy the folders `common`, `queue`, and `trace` from the `$AGENTJ/doc/ns2buildfiles/ns234` folder to the ns-allinone-2.34/ns-2.34 folder.

Note: Steps 5-9 are only necessary if you did not replace the Makefile.in of ns-2 as explained in step 2, but rather prefer to patch the Makefile manually.

Step 5:

Add the following variables to your Makefile.in file (near the top, just after you inserted the Protolib variables):

```
AGENTJ_SRC = $(AGENTJ)/core/src/main/c
AGENTJ_LIB_DIR = $(AGENTJ)/core/lib
AGENTJ_C_SRC = $(AGENTJ_SRC)/agentj
```

```
AGENTJ_UTILS = $(AGENTJ_SRC)/utils
AGENTJ_JNI = $(AGENTJ_SRC)/jni

AGENTJ_INCLUDES = -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/linux \
     -I$(AGENTJ_C_SRC) -I$(AGENTJ_UTILS) -I$(AGENTJ_JNI) -I$(AGENTJ_JNI)/jniheaders

AGENTJ_LIB = -L$(JAVA_HOME)/jre/lib/i386/server/ -L$(JAVA_HOME)/jre/lib/i386/ \
      -L$(JAVA_HOME)/jre/lib/amd64 -L$(JAVA_HOME)/jre/lib/amd64/server/ -ljvm
AGENTJ_SHARED_LDFLAGS = -shared -fPIC

OBJ_AGENTJ_CPP = $(AGENTJ_UTILS)/LinkedList.o \
    $(AGENTJ_C_SRC)/AgentjVirtualMachine.o $(AGENTJ_C_SRC)/Agentj.o \
    $(AGENTJ_JNI)/SocketWrapper.o $(AGENTJ_JNI)/JAVMSocketImp.o\
    $(AGENTJ_JNI)/JAVMTcpSocket.o $(AGENTJ_JNI)/JAVMDatagramSocket.o\
    $(AGENTJ_JNI)/TimerWrapper.o $(AGENTJ_JNI)/JAVMTimer.o \
    $(AGENTJ_C_SRC)/AgentJRouter.o \
    $(AGENTJ_C_SRC)/Ns2MobileNode.o
```

Step 6:

Provide paths to the AGENTJ include files by adding AGENTJ_INCLUDES to the "INCLUDES" macro already defined in the ns "Makefile.in", for example:

```
INCLUDES = \
-I. \
@V_INCLUDES@ \
-I./tcp -I./sctp -I./common -I./link -I./queue \
-I./adc -I./apps -I./mac -I./mobile -I./trace \
-I./routing -I./tools -I./classifier -I./mcast \
-I./diffusion3/lib/main -I./diffusion3/lib \
-I./diffusion3/lib/nr -I./diffusion3/ns \
-I./diffusion3/filter_core -I./asim/ -I./qs \
-I./diffserv -I./satellite \
-I./wpan \
$(AGENTJ_INCLUDES)
```

Step 7:

Add the list of AGENTJ object files to get compiled and linked during the ns build. For example, modify Makefile.in lines to the following

```
SRC = $(OBJ_C:.o=.c) $(OBJ_CC:.o=.cc) $(OBJ_AGENTJ_CPP:.o=.cpp) \
$(OBJ_EMULATE_C:.o=.c) $(OBJ_EMULATE_CC:.o=.cc) \
common/tclAppInit.cc common/tkAppInit.cc

OBJ = $(OBJ_C) $(OBJ_CC) $(OBJ_GEN) $(OBJ_COMPAT) $(OBJ_AGENTJ_CPP)
```

Step 8:

Add the rule for .cpp files to ns-2 "Makefile.in" (**NOTE** this has already been done - if you have installed protolib correctly):

```
.cpp.o:
    @rm -f $@
    $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $*.cpp
```

```
 and add to the ns-2 Makefile.in ‘‘SRC’’ macro definition:

    $(OBJ_CPP:.o=.cpp)
```

Step 9:

Create a shared library - define compile-time SHARED Library flags and libraries needed for your platform to create a shared library (this is needed for the JNI binding). On Mac OS 10.x, these are defined as follows:

```
AGENTJ_LIB = -framework JavaVM
AGENTJ_SHARED_LDFLAGS = -dynamiclib -lresolv
```

And the new rule for making the shared library should look like this:

```
libagentj.jnilib: $(OBJ) common/tclAppInit.o
$(LINK) $(AGENTJ_SHARED_LDFLAGS) -o $@ common/tclAppInit.o $(OBJ) $(LIB)
mv libagentj.jnilib $(AGENTJ_LIB_DIR)
```

or for Linux, these flags are defined as follows:

```
AGENTJ_LIB = -L$(JAVA_HOME)/jre/lib/i386/server/ -L$(JAVA_HOME)/jre/lib/i386/ \
-L$(JAVA_HOME)/jre/lib/amd64 -L$(JAVA_HOME)/jre/lib/amd64/server/ -ljvm
AGENTJ_SHARED_LDFLAGS = -shared
```

and adding AGENTJ_LIB to the “LIB” macro already defined in the ns “Makefile.in”, e.g.:

```
    LIB = \
@V_LIBS@ \
$(AGENTJ_LIB) \
@V_LIB@ \
-lm @LIBS@
```

and adding a new rule to make the shared library and put it in the correct place. For Linux, do:

```
libagentj.so: $(OBJ) common/tclAppInit.o
$(LINK) $(AGENTJ_SHARED_LDFLAGS) -o $@ common/tclAppInit.o $(OBJ) $(LIB)
mv libagentj.so $(AGENTJ_LIB_DIR)
```

For Linux, you also need to change the build rule for NS. This should be right under the line

```
# default for all systems but cygwin
```

Replace the build rule by:

```
NS_CPPFLAGS = -DNSLIBNAME=\"libagentj.so\"
NS_LIBS =  -ldl


$(NS): libagentj.so common/main-modular.cc
        $(LINK) $(NS_CPPFLAGS) $(LDFLAGS) $(LDOUT)$@ \
common/main-modular.cc $(NS_LIBS)
```

Step 9:
Run

```
./configure
```

in the ns source directory to create a new Makefile

Step 10:
Type:

```
make
```

to rebuild ns - this creates the static library

Step 11:
If you are using Mac, for compiling the Native Code, do not forget to type:

```
make libagentj.jnilib
```

to make the dynamic library needed for the installation of the JNI frameworks.

2.2 Installing the Java Code

Java has one dependency if you choose to command-line install:

1. **Apache Maven:** for compiling the Java classes we use Apache Maven. Maven can be found at the apache Web site *http://maven.apache.org/* or Google maven if you're feeling lucky.

And, so after the C++ part... finally, to compile the Java code, you do the following:

```
cd $AGENTJ
mvn install
```

2.3 Additional installation steps for using AgentJ as a routing protocol

If you want to use your Java routing protocol within AgentJ[1], you also have to copy some files into the Ns2 directory.

```
cp -r $AGENTJ/doc/ns2buildfiles/ns234/common $AGENTJ/../
cp -r $AGENTJ/doc/ns2buildfiles/ns234/trace  $AGENTJ/../
cp -r $AGENTJ/doc/ns2buildfiles/ns234/queue  $AGENTJ/../
cp -r $AGENTJ/doc/ns2buildfiles/ns234/tcl    $AGENTJ/../
```

This allows to add AgentJ as routing protocol to AgentJ, and to trace packets that are sent using a Java routing protocol.

2.4 Special instructions for 64-bit Linux operating systems

If you get the following error

```
relocation R_X86_64_32 against ‘a local symbol’ can not be used
when making a shared object; recompile with -fPIC
```

you need to add the -fPIC flag to all CFLAGS and LDFLAGS in the Makefile. If you followed the instruction in section 2.1 and replaced the Makefile.in, ns-2.34 should already be patched. However, you also need to add -fPIC in the Makefile.in files of tclcl and otcl (in the nsallinone-2.33 directory).

[1]Description of how to do so will soon be added in the manual

Chapter 3

AgentJ: The Toolkit

The chapter provides a brief snapshot of the AgentJ toolkit. It provides an overview of the AgentJ source tree, followed by a high-level overview of the software architecture. We then provide an explanation of what is required in order to create a Java Ns-2 Agent for AgentJ.

3.1 A Root Around the AgentJ Directories

AgentJ is made up of a collection of C++ and Java classes. The AgentJ directorty tree is organized as if it was a Java application. Therefore, within this AgentJ directory, there is a classes directory (where all classes live), a lib directory (for JAR files plus native shared libraries), a doc directory (containing this manual) and a src directory (for all source files), amongst others.

The src directory has a java and c subdirectory containing the Java and native code, respectively as shown in Figure 3.1. In the C directory there are three directories: agentj, javm and utils. The *agentj* directory contains the Ns-2 agent code (AgentJ.cpp) that implements the core Ns-2 agent capabilities for spawning the AgentJ software and tools. This class instantiates a *AgentjVirtualMachine* object, also contained in this directory, upon first creation, which creates a Java virtual machine for interacting with the Java codes. It thereafter passes commands via the *AgentjVirtualMachine* class to be passed to the Java objects (using a pointer to the Ns-2 C++ agent for identification). The *javm* directory contains the classes for the implementation of the Java native interface Java Virtual Machine (JAVM), which bind to the underlying Protolib classes for implementation for integration into Ns-2.

In the Java directory, we have the corresponding *javm* directory that contains the Java side of the AgentJ implementation of *java.net*, *java.io* and some *java.lang* clases. It also contains the reciprocal of the C++ AgentjVirtualMachine class, called AgentJVirtualMachine.java, which contains the entry point static methods called by the C++ class and the *AgentJNode.java* class which is the abstract base class implementation of any Java NS-2 agent implemented by a user. In the root directory, there is also a NAMCommands class that allows you to annotate messages and colors onto your NAM visualisation during execution. There is an *examples* directory, which is covered in chapter 4 and a *thread* directory which contains the Java code for thread synchronisation and the monitoring of threads during execution.

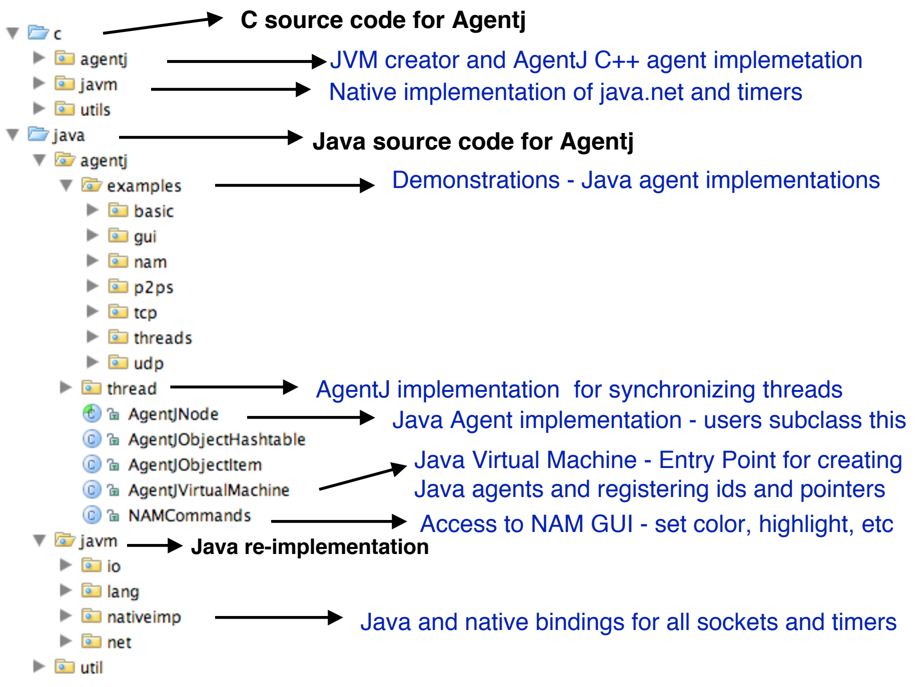

c
C source code for Agentj
agentj
JVM creator and AgentJ C++ agent implemetation
javm
Native implementation of java.net and timers
utils
java
Java source code for Agentj
agentj
examples
Demonstrations - Java agent implementations
basic
gui
nam
p2ps
tcp
threads
udp
thread
AgentJ implementation for synchronizing threads
AgentJNode
Java Agent implementation - users subclass this
AgentJObjectHashtable
AgentJObjectItem
AgentJVirtualMachine
Java Virtual Machine - Entry Point for creating Java agents and registering ids and pointers
NAMCommands
Access to NAM GUI - set color, highlight, etc
javm
Java re-implementation
io
lang
nativeimp
Java and native bindings for all sockets and timers
net
util

Figure 3.1: The C++ Classes within AgentJ.

3.2 From the Java Side to the C Side

Since AgentJ is a Java environment for Ns-2, a JNI bridge is needed in order to map between the C++ NS2 agents and the associated Java agents. This bridging mechanism is required at both the input to the Java initialisation and at its lower communication layers; that is, first the C++ NS-2 agents need to instantiate and attach Java agents to the NS-2 C++ agents, then the Java agents need to be able to access the Protolib interface in order to pass data between themselves in NS-2. In the first case, a C++ environment creates a Java Virtual Machine (JVM) (using the *AgentjVirtualMachine.cpp* and *AgentJVirtualMachine.java* classes) for creating and accessing Java objects. In the second case, we provide a mapping from *java.net* to the Protolib C++ libraries using JNI. The resulting interaction is shown in Figure 3.2. This figure shows how Java is used by NS-2 to create the overall picture. Each C++ NS2 *agent* attaches a Java object (i.e. Java agent) via a call to *AgentjVirtualMachine.cpp* which invokes the *attachJavaAgent* method in *AgentJVirtualMachine.java* using JNI for registration of the agent.

There could potentially be thousands of NS2 nodes and each one might want to instantiate and use a Java object. Therefore serious scalability issues can be encountered if this interaction is not slimline enough. In AgentJ, the C++ JVM helper class (AgentjVirtualMachine.cpp) therefore only creates one JVM per simulation. The JVM then interfaces through a singleton *AgentJVirtualMachine.java* class, which creates and manages the Java objects. *AgentjVirtualMachine* contains functionality that can dynamically create a Java object from a textual representation of its name (e.g. agentj.examples.tcp.SimpleTcp). The registration request (using *attachJavaAgent*) results in a Java *Hashtable* being populated with an item pair; the C++ NS-2 agent pointer representing the key and AgentJ Java object representing the object.

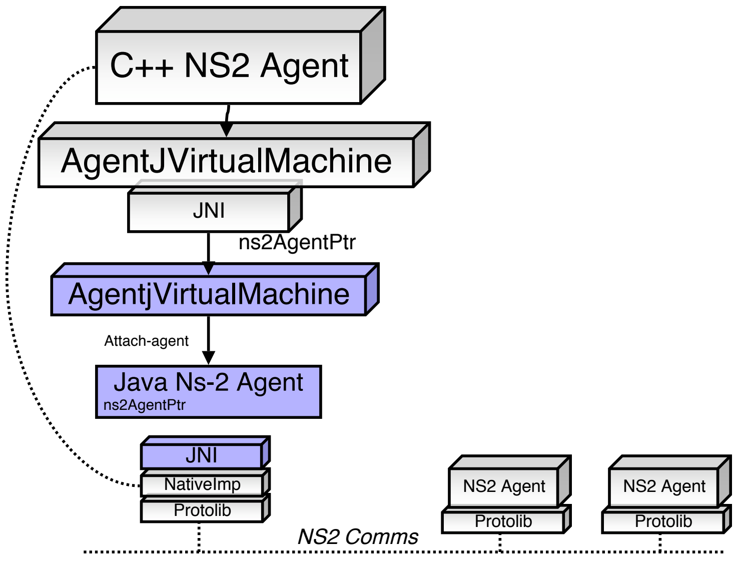


Figure 3.2: An overview of the AgentJ software components

The C++ NS2 agent's *ID* is actually its C++ pointer, which is reused later within the JNI binding to bind the sockets and timers onto the Ns-2 agent that issued the command to attach them.

3.3 Sending commands to Java Agents

A Java NS2 agent is attached to a C++ NS2 node and is accessed in the same way as C++ Ns2 agents are accessed, through the command() interface. Thus, specifically, an AgentJ node is:

> *A Java object that extends the* AgentJObject *abstract class (and therefore implements the required command() method).*

Ns-2 Java agents typically provide commands that hook into 3rd party Java application in order to provide simulation tests for those applications. You can think of Ns-2 agents as a replacement to a GUI or a command line interface that instructions your application to do certain things at a certain time with respect to the choreography of your simulation i.e. to specify what you want to simulate.

Figure 3.3 shows interaction between the simulation orchestration (in TCL), the C++ agent and its associated Java class. The programmer who wishes to use Java functionality within their NS2 simulations only needs to be concerned within their NS2 TCL script and their Java class that implements the behaviour they require. The relationship between an Ns2 agent and its Java class is very similar to the relationship between an NS2 TCL script and its associated C++ class (i.e. an NS2 agent) which implements the same kind of interaction through sending text commands between the two. The Java interface employs the same mechanism to bridge these different programming languages. The C++ agent (Agentj.cpp) simply acts as a go-between to pass commands across to the appropriate Java

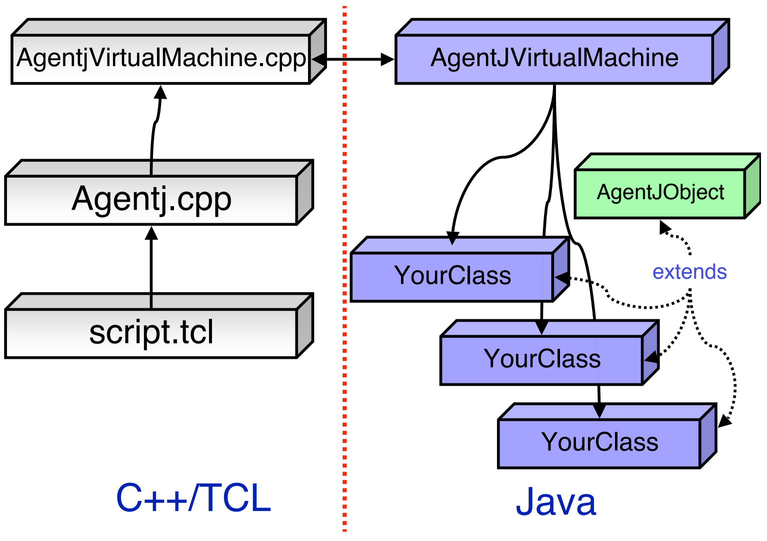


Figure 3.3: The interaction between TCL and Java objects through the command() method in AgentJObject.

object, via the C++ AgentjVirtualMachine and corresponding AgentJVirtualMachine Java classes.

The command-style interface satisfies some essential AgentJ design conditions:

- **Simplicity:** the scalability issues and framework for interacting with the Java objects can easily be hidden behind the container C++ and Java classes - the programmer does not need to be aware of their presence.
- **Familiarity:** this mechanism allows communication between the NS2 agent and any attached Java class through the same familiar interface as NS2 programmers interface between the TCL scripts and C++ agents now. Users and programmer do not need to learn a new mechanism.

This interaction is shown in Figure 3.4, which illustrates the two key commands for attaching and interacting with a Java agent from TCL. Each Ns2 node creates a Java object of its own choice by using the TCL command:

```
attach-agentj <class>
```

where class can be any fully qualified Java class name. For example, to attach the ChangeDelimiter class to a node, you would use:

```
attach-agentj agentj.examples.basic.ChangeDelimiter
```

Once the Java object has been created, commands can be sent by using the TCL command:

```
agentj <command> <args>
```

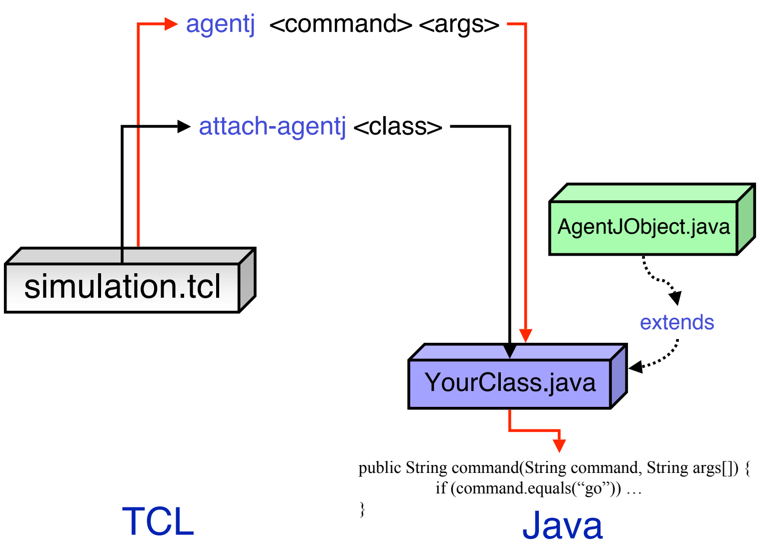


Figure 3.4: The interface to a Java program for an agent employs a similar interface to that of NS2 when communicating between the TCL scripts and the C++ classes.

which would invoke the java command with the associated arguments. So, to invoke the command "hello" on ChangeDelimiter, one would use:

```
$ns_ at 0.0 "$a1 agentj hello A-String-From-P2"
```

which would have a corresponding implementation within the ChangeDelimiter object, like the following:

```
    public boolean command(String command, String args[]) {
       if (command.equals("hello")) {
            System.out.println("ChangeDelimiter(" + myID + ") has "
                    + args.length + " arguments");
            for (int i=0; i<args.length; ++i) {
                System.out.println("Arg[" + i + "] = " + args[i]);
            }
            return true;
        }
       return false;
    }
```

You can try this example out yourself by running the following:

```
cd $AGENTJ/examples/basic/
ns changeDelimiter.tcl
```

which should show a number of set-up commands and have an output ending in:

```
Proto Info: AGENTJ COMMAND: hello, Arguments = A-String-From-P2
changeDelimiter(2) has 4 arguments
Arg[0] = A
```

```
Arg[1] = String
Arg[2] = From
Arg[3] = P2
```

This process is demonstrated further in the next chapter where a number of more complex examples are provided.

Chapter 4

AgentJ Examples

The AgentJ example TCL scripts for running within Ns-2 can be found in the *examples* directory within the AgentJ route. The corresponding Java source code that implements these demos can be found in the *src/java/agentj/examples* directory. There are several demonstration **directories** that contain the various demos AgentJ has to offer. They are:

basic: Contains some basic demos that show how agents are created and how commands are invoked on those agents. This directory for example contains the change delimiter demo described in the previous chapter and it also contains some demos on how to use the timers to schedule timeout triggers at certain points during the simulation.

udp: Contains a number of demos using UDP, ranging from simple client and server demos, to using multicast groups, to sending messages to multiple nodes.

tcp: Contains some demos for creating TCP client and server sockets. The demo simpletcp.tcl is described in more detail below. This directory also contains a demo that combines multicast (for discovery of the nodes) and TCP for transmission of a message (see multicastAndTcp.tcl).

threads: This directory contains demos that create Java threads for implementing the conventional way that a Java application would create a non-blocking receive() on a socket. The demostrations show that an application can spawn multiple threads and that AgentJ keeps a track of such threads and monitors their lifetime. One demo also shows the use of creating multiple socket threads and timers to trigger the points at which the application sends the messages, implemented using the java Thread.sleep() method.

nam: The NAM application is a visual animator for NS-2 simulations. Typically, within the TCL script, you would insert commands in order to display labels or change colours of the nodes during the simulation. The namdemo.tcl example here shows how to do this from within your Java application itself. This demo is described in Section 4.2

gui: Contains a demo that implements a graphical user interface on an Ns-2 node to poll for input. This demonstrates the capability of allow user interactions within an Ns-2 simulation.

In each directory, there is a README.TXT file, which provides an overview of each demo in that directory. The examples, described here provide a high-level illustration of just a few of the examples. It assumes that you have some knowledge of running NS-2 simulations and that you will try these demos as you read this manual for a complete picture of the TCL and Java parts of the examples. If you are not familiar with NS-2, you will need to at least try running a few Ns-2 examples before attempting these.

4.1 TCP Socket Example

In the *tcp* directory, there contains a simple demo that implements the Java socket example in the Java tutorial. This demonstrates how a programmer can use the blocking Java API with the sequential nature of the NS-2 simulator. AgentJ detects all blocking calls and creates a behind the scene non-blocking callback for such calls, which wait for data to arrive before releasing the blocked code. Hence, a receve() call on a socket, to the Java application, simply waits (blocks) until data arrives. In AgentJ however, the call does not block, it releases control back to NS-2 and passes the control to the next node in the simulation, and waits for data asynchronously in the background.

4.1.1 TCP Socket Example: The TCL Side

The TCL file "simpletcp.tcl" can be found in the *examples/tcp* directory. It simply creates two nodes with a drop tail 64kb network link connecting them. It then creates the two Java agents and invokes commands on those agents as follows:

```
$ns_ at 0.0 "$p1 attach-agentj agentj.examples.tcp.SimpleTCPServer"
$ns_ at 0.0 "$p2 attach-agentj agentj.examples.tcp.SimpleTCPClient"

$ns_ at 1.0 "$p1 agentj open"
$ns_ at 2.0 "$p1 agentj accept"

$ns_ at 6.0 "$p2 agentj open"
$ns_ at 10.0 "$p2 agentj send"
```

Note how the "attach-agentj" command, described in the previous chapter is used to attach the agents. You can attach any Java object that extends *AgentJNode* to an Ns-2 node using this command. You can then pass commands to those objects as illustrated here. In this example, we tell the server (p1) to open a server socket and then to call the accept method on that socket to wait for an incoming connection.

We then tell the client to open a socket and connect to the server and then send data to the server socket. The result of the simulations will be something like:

```
SimpleTCPServer, receing Message !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--> Hello server, are you working?
```

```
SimpleTCPServer:: received hello message from client, sending a Reply !!!!!!!
SimpleTCPClient, receing Message !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--> Ah, Hello client, nice to make your acquaintance...
```

As shown above, when the server socket receives the incoming message, it sends a reply to the client socket indicating that the send was successful. The client socket receives this message and the simulation ends. This demo is simple but shows the power of how AgentJ can run unaltered Java code. The Java code here is trimmed replication of the Java tutorial and contains exactly the same code one would need to implement this over real world sockets. In fact, the same Java agent code can be run outside of AgentJ to show this demo running over a real network using the Java runtime.

4.1.2 TCP Socket Example: The Java Side

On the java side, we have two classes:

- "agentj.examples.tcp.SimpleTCPServer" and
- "agentj.examples.tcp.SimpleTCPClient"

found in the *agentj/src/java/agentj/examples/tcp* directory.

You can view source code directly but the example is equivalent to the Sun Java Tutorial Knock Knock example for creating a TCP socket connection that can be found at:

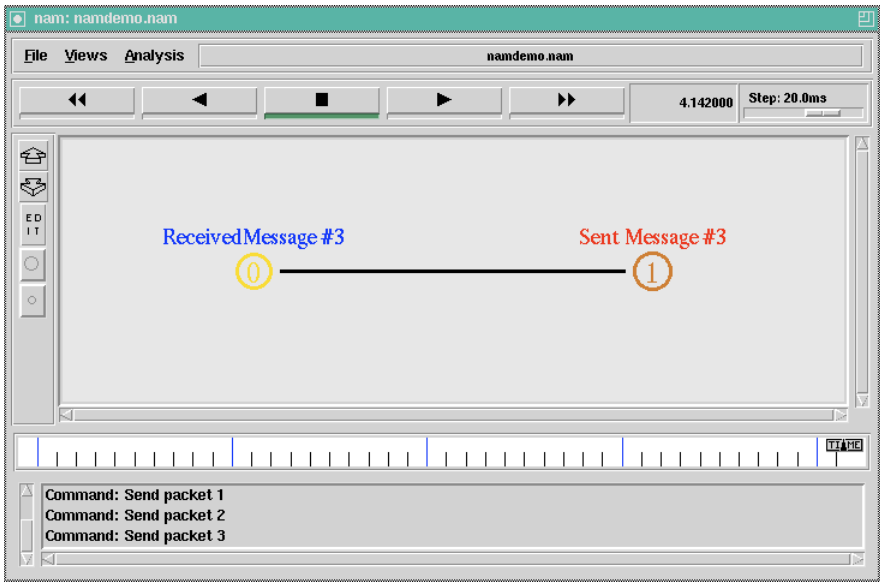
```
http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html
```

Note that in this source code, the agents simply import the java.net.* classes and write to those APIs. This is normal for AgentJ but behind the scenes these classes are bytecode rewritten to use the AgentJ implementations of these classes. With respect to AgentJ, the only functional difference between the Java tutorial example and the code here is to set the logging level for the Java code and C++ code. This is set in the constructor as follows:

```
setJavaDebugLevel(Level.ERROR);
setNativeDebugLevel(AgentJDebugLevel.error);
```

which indicates that only errors should be output to the stdout display. An agent can set this level to whatever it choses.

AgentJ has two logging systems, one for Java and one for the C++ code. The Java logging uses the Log4J system [?] and the C++ code uses the Protolib logging mechanism, PLOG. The logging levels available for each are defined in the "org.apache.log4j.Level" and "AgentJNode" classes respectively.



```
Command: Send packet 1
Command: Send packet 2
Command: Send packet 3
```

Figure 4.1: The Output NAM display for the namdemo.tcl simulation

4.2 Animating NAM Demo

This demonstration shows how add NAM display instructions from Java. Within your Java code directly, you can change the colors of nodes, add markers, add labels and add trace messages to the NAM animations and AgentJ inserts the commands at the right point in time in the nam trace file for visualisation. This provides a very powerful annotation mechanism for simulations because often there are several message exchanges before the control comes back to TCL, so clearly annotating such messages in TCL is not enough. Using the AgentJ NAM interface, each programmatical step can be annotated within the NAM simulation.

4.2.1 NAM Demo: The TCL Side

This example can be found in *examples/man/namdemo.tcl*. This example also shows how to attach the agents and pass commands in order to orchestrate the simulation.

```
$ns_ at 0.0 "$p1 attach-agentj agentj.examples.nam.NamDemo"
$ns_ at 0.0 "$p2 attach-agentj agentj.examples.nam.NamDemo"

$ns_ at 0.0 "$p1 agentj init-server"
$ns_ at 0.0 "$p2 agentj init-client"

$ns_ at 1.0 "$p1 agentj receive"
$ns_ at 1.0 "$p2 agentj send"
```

The receive and send commands are repeated four times. The resulting NAM display can be seen in Figure 4.1, which shows the nodes labelled with the current message displayed in different colors.

4.2.2 NAM Demo: The Java Side

The demo actually demonstrates how to color nodes and add labels during a simulation. We use a simple multicast demo to ilustrate this. Every time a message is sent it is added as a label to the NAM animation and colors are changed. The commands are shown in the *send* and *receive* methods below from the NAMDemo class:

```
NAMCommands nam;

 public NamDemo() {
     this.setJavaDebugLevel(Level.DEBUG);
     this.setNativeDebugLevel(AgentJDebugLevel.detail);
     nam = this.getNamCommands();
     nam.setAnimationRate(0.02);
 }


public void send() {
     nam.setNodeColor(NAMCommands.NamColor.chocolate);
     try {
         String msg = "Message #" + msgcount;
         ++msgcount;
         DatagramPacket hi = new DatagramPacket(msg.getBytes(),msg.length(), group);
         nam.setNodeLabel("Sent " + msg, NAMCommands.LabelPosition.down,
          NAMCommands.NamColor.red);
         s.send(hi);
     } catch (IOException e) {
         e.printStackTrace();
     }
 }


 public void receive() {
     nam.setNodeColor(NAMCommands.NamColor.gold);
     byte[] buf = new byte[1000];
     DatagramPacket recv = new DatagramPacket(buf, buf.length);
     try {
         s.receive(recv);
         String msg = new String(recv.getData());
         nam.setNodeLabel("Received" + msg, NAMCommands.LabelPosition.up,
          NAMCommands.NamColor.blue);
     } catch (IOException e) {
         e.printStackTrace();
     }
 }


if (command.equals("send")) {
    nam.traceAnnotate("Command: Send packet " + msgcount);
    send();
```