AgentJ: Enabling Java NS-2 Simulations for Large Scale Distributed Multimedia Applications

Ian Taylor
School of Computer Science
Cardiff University and
Center for Computation and Technology
Louisiana State University
email: Ian.J.Taylor@cs.cardiff.ac.uk

Brian Adamson, Ian Downard, Joe Macker
PROTocol Engineering and
Advanced Networking (PROTEAN) Research Group
United States Naval Research Lab (NRL)
Washington, DC
email: ian.downard@nrl.navy.mil

Abstract

We present a framework, called AgentJ that extends the NS-2 platform to support the simulation and performance analysis of Java network applications. We particularly focus on the simulation of P2P networks and have already integrated the P2PS middleware for the simulation of super-peer networks for discovering participants in large scale Internet applications, which has a high significance for distributed multimedia applications to enable overlays for the discovery and searching across large numbers of distributed multimedia resources. AgentJ builds upon the numerous years of networking research with NS-2 and leverages the Protolib toolkit from NRL to facilitate the passing of real data between NS-2 nodes, thereby creating a platform for the simulation of content-based middleware and applications. AgentJ currently supports UDP unicast and multicast and maintains compatible programming interfaces with the standard Java network package as well as the NS-2 scripting interfaces employed for staging simulations.

I. Introduction

AgentJ provides the unique ability to simulate real-world Java applications within the NS-2 Network Simulator [1]. It supports transport protocols (e.g. UDP unicast and multicast) and timer utilities, which are commonly used in real-world network applications and simulations. The purpose of AgentJ is to extend the NS-2 network simulator with the capability to simulate unmodified Java network applications. For application developers, AgentJ provides a distributed simulation environment for Java applications for performance analysis and optimization.

One of the main focuses for AgentJ is to facilitate the simulation of Java peer-to-peer (P2P) architectures, such as P2PS [2] and Jxta [3], which allow the composition of large scale networks with transient connectivity and access from users on home computers and devices. To this end, our main driver for this integration has been the P2PS middleware, which offers numerous constructs for programming within ad-hoc P2P networks, such as rendezvous peers (for creating centralized-decentralized or super-peer networks), grouping (for security and scoping), virtual pipes (that have bindings to several transport protocols) and several discovery mechanisms. The discussion of this integration is beyond the bounds of this paper but extensive documentation can be found on AgentJ Web site [4]. Here in this paper, we focus on the underlying enabling framework, AgentJ.

Such systems, like P2PS, allow the direct simulation of discovery and communication mechanisms for large scale P2P networks. Both Jxta and P2PS create virtual overlays [5] that allow the construction of scalable unstructured networks. By building upon early P2P networks, such as Gnutella [6], both Jxta and P2PS extend and generalize on this idea to create dynamic ad-hoc overlays typically constructed using caching peers and application-level routing similar to those implemented in file-sharing networks, such as Limewire [7] and KaZaA [8]. Another example of such an application is Skype [9], which is capable of decentralized (P2P) discovery (using super peers) of its network participants for connectivity for voice and video conferencing. We believe such distributed architectures and network overlays will become important for distributed multimedia applications, where searching for participants, or media by indexing or even by content, will become a necessary part of the application process. We envisage therefore that AgentJ could play an important role here in such research by providing a realistic simulation testbed to allow rigorous testing of such P2P protocols and algorithms across ubiquitous networks before deployment, something lacking in today's development to production software life cycle for distributed Java applications.

The rest of this paper is organized as follows. The next three sections provide the background for the AgentJ project, related work within this field and a brief description of the underlying technology, NS-2, providing a high-level vision for the AgentJ software. In Section II, we provide an overview of the AgentJ architecture, which is expanded in detail in the following two sections, describing the underlying interfaces and the software stack, respectively. It is out of the bounds of this paper to provide source code examples but we have compiled a number of these on the Web site [4], including examples integrating the P2PS middleware, which involve several complex distributed discovery and communication scenarios.

A. AgentJ Background

AgentJ has been developed within the Scalable Robust Self-organizing Sensor (SRSS) network project at the United States Naval Research Laboratory (NRL) [4]. This project is an effort to build network protocols for organizing heterogeneous sensor devices in an inherently unreliable and disadvantaged Mobile Ad-hoc Network (MANET) [10]. A portion of this project is dedicated to developing lightweight peer-to-peer protocols for resource discovery. It is anticipated that P2P middleware will facilitate autonomous self-organization of sensor nodes and their various related functions for data collection, processing, and reporting. To satisfy these goals, the SRSS project has been utilizing AgentJ to evaluate lightweight peer-to-peer (P2P) solutions for dynamically discovering and connecting mobile nodes, including Jxta [3] and then more recently the P2PS middleware [2].

B. Related Work

The Java Network Simulator [11] is a Java implementation of the NS-2 simulator. Just like NS-2, it provides developers access to the simulation of networking protocols and then produces a trace file (same format as NAM trace files) which can be viewed in a network animator such as Javis [12]. At the time of writing, version 1.7., supports multicasting, multithreading and dynamic scheduling. JNS also supports the simulation of ”real” Java network applications by providing a fake.net.MulticastSocket and a DynamicScheduler that can transform network calls made from a Java program into schedulable events on JNS.

J-Sim[13] is a real-time process-based simulation environment written entirely in Java which provides a script interface with Perl, Tcl or Python. J-Sim defines a each entity in a network stack according to an autonomous component-based architecture. This design emphasises modularity and promotes object-oriented code reuse.

In many ways, JNS and J-Sim provide similar environments to AgentJ. However, they only support a fraction of the transport protocols and environments that the original NS-2 framework implements. AgentJ was designed to operate as a thin layer on top of this framework, which is advantageous not only because it inherits many protocols already supported by NS-2, but also because it can be extended to new protocols by creating an interface, rather than reimplementing each one natively in Java. Furthermore, JNS and J-Sim both lack the flexibility to run the same C++ implementations of network protocols which are used in physical networks. NS-2 is desireable in this respect, because it supports the Protolib toolkit [14], which in turn supports simulating the same C++ source code used for network protocols in the real world.

C. An NS-2 Overview

NS-2 [1] is a discrete event simulator targeted at networking research, and has been an on-going project for the past sixteen years. NS-2 has been developed considerably during this timeframe and therefore contains unparalleled support in the open-source domain for a wide range of transport and routing protocols over both wired and wireless networks. An international community of users and developers in commercial, academic, and government institutions use NS-2 as a major tool for researching a wide range of topics in computer networking. Some of this research has expanded the capabilities of NS-2, and includes publicly available extensions for MANET routing, wireless communications, sensor networking, peer-to-peer networking, and various tools for post-processing simulation results.

Fundamentally, NS-2 is a discrete-event simulator for modeling packet-based computer networks. It’s framework is modular in nature, which facilitates extendability. It consists of components in object-oriented Tcl (oTcl) and C++. The oTcl portions are designed to provide a user interface to the simulation executive where simulations can be setup in a scriptable manner without the need for recompiling. Typically, simulations using the existing NS-2 modules are setup with Tcl scripts, where nodes, data flows, applications, underlying networking protocols, as well as every other aspect of a network can be configured. The C++ portions are designed to maximize run-time speed of the NS-2 executive and its network models. An overview of this architecture is illustrated in Figure 1.

D. The NS-2 Simulator and AgentJ

AgentJ provides NS-2 users with the capability of simulating real-world Java network applications. It does this in two ways. First it creates an interface for users to control Java applications from TCL scripts. This interface hooks the user’s control capabilities through the three different languages, namely oTCL to C++ to Java (see Section III-A for more details). Second, it instantiates within NS-2 a single Java Virtual Machine (JVM) which interprets the Java applications and tracks their state on behalf of the NS-2 C++ executive. This is achieved by extending the core NS-2 framework to provide an interface to Java code for each node in the distributed simulation.

Although the implementation of AgentJ within the NS-2 framework is very complex, it is designed to have a nonintrusive impact on users of NS-2 and developers of Java network applications. In this way, the simulation of Java applications in NS-2 follows the same conventions as the simulation of traditional C++ applications for NS-2, and the development of Java network applications follows the same conventions as the standard *java.net* classes. This seamless interface for Java in NS-2 is described in more detail in Section III-A.

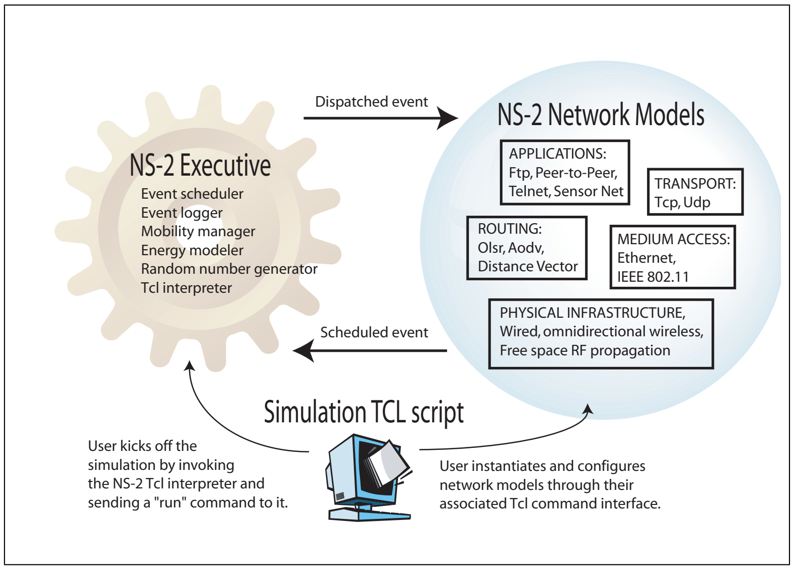

Dispatched event
NS-2 Network Models
NS-2 Executive
Event scheduler
Event logger
Mobility manager
Energy modeler
Random number generator
Tcl interpreter
APPLICATIONS: Ftp, Peer-to-Peer, Telnet, Sensor Net
TRANSPORT: Tcp, Udp
ROUTING: Olsr, Aodv, Distance Vector
MEDIUM ACCESS: Ethernet, IEEE 802.11
PHYSICAL INFRASTRUCTURE, Wired, omnidirectional wireless, Free space RF propagation
Scheduled event
Simulation TCL script
User kicks off the simulation by invoking the NS-2 Tcl interpreter and sending a "run" command to it.
User instantiates and configures network models through their associated Tcl command interface.

Fig. 1. The NS-2 executive includes modules for handling event schedules, random number generation, node mobility, energy constraints, and event tracing to log files. The NS-2 network models can cover any aspect of a layered network architecture, such as physical layer access control, link layer queuing, network layer routing, or application layer services. NS-2 provides an object-oriented Tcl interface for users to setup simulations and customize network models.

II. Overview of the AgentJ Architecture

AgentJ is built on top of the NS-2 C++ agent framework. Therefore, it also supports C++ via defined interfaces (using the PAI or Protolib toolkits, see section II), which allow interaction with the lower-level transport protocols. However, in AgentJ's Java implementation, care has been taken to provide interfaces which use the same class structure as a Java programmer would use to program sockets (i.e. the implementations in the *java.net* package). In order to facilitate simulating Java code which creates standard *java.net* sockets for real-word networking, AgentJ uses on-the-fly Java bytecode rewriting to convert socket implementations for NS-2 compatibility. However, since NS-2 is a discrete event simulator, common coding techniques, such as blocking on a method or the use of threads prohibits simple integration. Within AgentJ therefore, we provide some alternative thread implementations for common networking requirements, such as event generation from sockets upon the receipt of data.

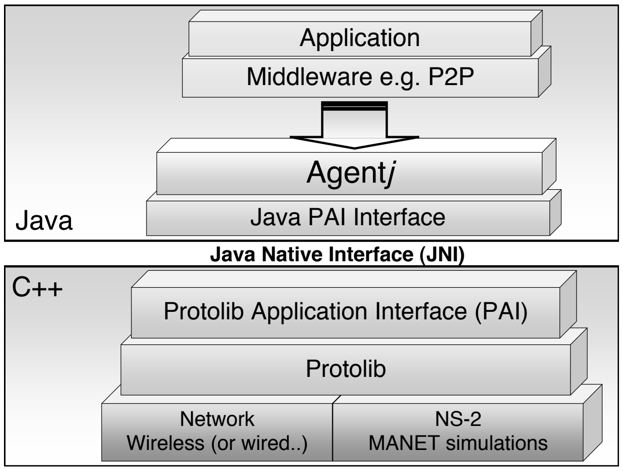

Application
Middleware e.g. P2P
Agentj
Java PAI Interface
Java
Java Native Interface (JNI)
C++
Protolib Application Interface (PAI)
Protolib
Network Wireless (or wired..)
NS-2 MANET simulations

Fig. 2. The AgentJ framework sits between the application layer and the physical layer. Its primary purpose is to support the simulation of Java network applications in NS-2.

AgentJ implements a framework for running Java applications, middleware, etc. in NS-2. We begin our explanation of AgentJ with an overview of this framework as illustrated in Figure 2. Note how the physical layer in Figure 2 is separated into a real-world IP environment, and an NS-2 simulation environment. NS-2 [1] is an open source project lead by the Information Sciences Institute at the University of Southern California. An international community of users and developers in commercial, academic, and government institutions use NS-2 as a major tool for researching a wide range of topics in computer networking. Fundamentally, NS-2 is a discrete-event simulator for modeling packet-based computer networks. It's framework is modular in nature, which facilitates extendability.

The toolkit which abstracts the physical layer between real world and simulated environments is called Protolib [14]. The Protean protocol prototyping library (Protolib) is a set of C++ classes which facilitate the development of platform-independent network software. It provides an abstracted interface to timers, sockets, and addresses for operating systems such as MacOS, Unix, and Windows, as well as simulation systems such as NS-2. Applications or network protocols which are implemented with Protolib can be run or simulated on any of these platforms without modifying the application's timer or networking functions. The actual binding of an application's Protolib components to their system dependent implementations is achieved at compile-time.

The PAI interface (see Section III) extends the flexibility of Protolib by allowing multiple listeners to be attached to event handlers for sockets or timers. It also provides convenient interfaces for creating and deleting socket or timer objects and implements an engine for providing the necessary housekeeping. In essence, PAI provides a hosting environment for Protolib to be deployed in Java applications because it fills in the gap between what Java applications expect and what the core Protolib toolkit provides.

Finally, AgentJ provides a familiar Java interface, based around the *java.net* package, to the underlying functionality provided by the PAI and Protolib interfaces. Specifically AgentJ implements a Java Native Interface (JNI) [15] binding for the PAI C++ interface and then re-implements this functionality within a set of Java classes that allow Java programmers to use Protolib sockets as if they were native Java ones. The next section describes this interaction in more detail.

III. AgentJ and the JNI PAI Interface

Since AgentJ is written in Java, a JNI bridge is used to map between the C++ objects of NS-2 and the associated Java objects of AgentJ. This bridging mechanism is also required at the input to user-level Java applications and their lower communication layers. In the first case, the C++ agents create a Java Virtual Machine (JVM) which interprets high-level Java commands from the Java NS-2 application. In the second case, Java timer and socket calls are mapped through the JNI and PAI to Protolib C++ libraries (see Figure 2. In this figure, we show the interactions between each of the components in the AgentJ system: the application developer uses the AgentJ Java network classes; these are converted to a set of Java PAI function calls; which are converted via the JNI interface to the C++ PAI interface; before being passed down to the Protolib interface and the underlying implementations to a NS-2 or a networked environment.

The Protolib Application Interface (PAI) provides a generic interface to timers, sockets and dispatchers and employs a factory-design approach in order to instantiate the required set of objects, such as UDP sockets, real-time timers, NS-2 timers and NS-2 event dispatchers.

The resulting C++ PAI interface looks very similar to a Java interface. Wherever possibly, we have used the Java conventions for interfacing with the underlying objects. For example, instead of providing callback functions for a timer, we allow a user to attach multiple listeners to a timer in order to get notified when an event occurs. The resulting interface therefore is simple and very Java friendly, making the JNI integration with the Java classes far easier to define and implement.

A. The User Interface to Simulated Applications

AgentJ was designed to include a "user interface" to Java applications in NS-2 which is consistent with the "user interface" to C++ applications in NS-2. User interactions are passed to simulated applications by scheduling commands in the simulation script, since it is not (yet) possible to have a GUI for each application on each simulated network node. In this sense, the "user interface" to NS-2 application agents is more appropriately referred to as a *command interface*. In this section, we describe this command interface, and show how AgentJ is consistent with that design.

NS-2 consists of components implemented in object-oriented Tcl (oTcl) and C++. The oTcl portions are designed to provide a user interface to the simulation executive where simulations can be setup in a scriptable manner without the need for recompiling. Typically, simulations using the existing NS-2 modules are setup with Tcl scripts, where nodes, data flows, applications, underlying networking protocols, as well as every other aspect of a network can be configured. The C++ portions are designed to maximize run-time speed of the NS-2 executive and its network models.

The oTcl and C++ components of NS-2 are linked together with an interface which follows the style of command-line arguments interpreted by `main(int argc, char **argv)` in C programs, where the argument count (`argc`) and argument vector (`argv`) parameters are used for passing command-line parameters. In NS-2, commands are issued not from a command line, but by oTcl scripts, and they are processed not by `main(int argc, char **argv)`, but by the command interpreter method of the C++ module being invoked.

Within the context of AgentJ, the mechanism whereby user commands are passed from TCL scripts to Java applications goes through the PAI C++ module. Namely, the PAI invokes a standard NS-2 method, called:

```
int PAIAgent::command(int argc, const char*const* argv) {
```

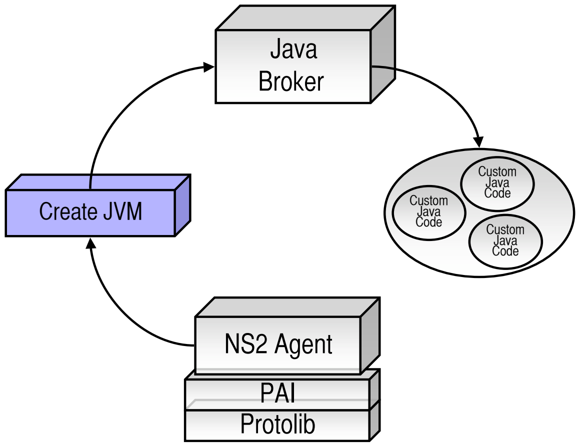


Fig. 3. AgentJ can attach any conforming Java object to an NS-2 agent.

which captures the user's command and forwards it through the JNI to the command interpreter in the Java application.

IV. AgentJ Software Overview

In this section, we describe how the AgentJ framework has been integrated into NS-2 using the JNI, the core C++ Protolib libraries, the PAI, and Java classes. Figure 3 provides a simple overview of how the software components are connected when creating and attaching Java objects to NS-2 C++ agents. As illustrated, each NS-2 C++ *agent* attaches a Java object (i.e. Java agent) by accessing a Java Virtual machine (JVM) and by requesting that an association be made within the Java domain between the desired Java object and itself. This request results in a Java *Hashtable* being populated in the *JavaBroker* with an item pair where the C++ agent is the key and AgentJ Java agent is the object.

There could potentially be thousands of NS-2 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 therefore, we only allow **one** JVM to be created no matter how many nodes exist in the simulation. This is achieved through the C++ JVM helper class (C2JBroker) shown in Figure 4. The JVM instantiates a singleton Java class called the *JavaBroker* (as illustrated), which creates and manages all Java objects in AgentJ. The *JavaBroker* contains functionality that can dynamically create a Java object from a textual representation of its class name and package (e.g. `examples.NS-2.SimpleCommand`).

When Java objects are created, they are added to a local Hashtable in the *JavaBroker*. This entity then maintains an association between the new Java object and the NS-2 agent from which it was derived. The NS-2 agent's *ID* is actually its C++ pointer, which is reused later within the JNI binding (see next section). We therefore create a one-to-one mapping between the C++ NS-2 agent in the JavaBroker class, which acts as a central locater for all Java objects and keeps memory allocation down to a minimum.

When a node wishes to send a message to its attached Java object, it must first locate this object by searching this Hashtable using the pointer to the C++ agent. Once it has obtained a reference to the Java agent, it can forward the command to the object. However, this fits in rather nicely with the discrete event dispatching scheme adopted by NS-2, which switches between nodes during execution – AgentJ switches at the same time and therefore also keeps the switching overhead to a minimum; that is, a number of C++ to Java invocations would only involve one lookup at the beginning of this agent's time slice.

A. Inter-AgentJ Communication

Once an NS-2 node has been configured with an AgentJ module, its Java network application can then use the AgentJ communication and timing facilities in order to schedule events and communicate with other nodes. These facilities can be accessed by the **standard Java interfaces** which have been refactored to bind to the simulation world within NS-2 (using PAI and Protolib as described earlier). AgentJ currently supports calls to the following standard Java interfaces:

- **UDP Transport:** DatagramSocket and MulticastSocket have been refactored to PAI and Protolib for functionality in NS-2. [1]
- **Inet Support:** a number of the InetAddress methods have been implemented to work within the NS-2 context.

[1]Extensions for Simplified Multicast Forwarding must be obtained in order to simulate multicast sockets with *wireless* networks in NS-2. These extensions are available from the Protean research group at NRL.

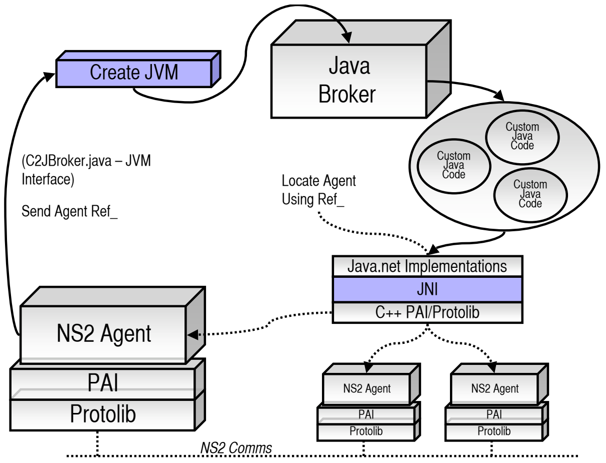


Fig. 4. An overview of the AgentJ software interaction between NS-2, JNI, PAI and Protolib.

- **Timing Libraries:** simple interfaces to timing functions have been implemented so that NS timing events can be triggered at specific intervals during the simulation.

These functions have been enabled for AgentJ by re-implementing their standard Java interfaces as PAI compatible interfaces. Therefore, in order to use these functions within NS-2, one would instruct the Java application to import the factory implementing the corresponding PAI implementation.

Figure 4 gives an overview of this integration and provides a complete picture of how AgentJ interacts within Java. As mentioned earlier, AgentJ not only creates a JVM to interact with Java objects but these Java objects, in turn, use JNI to interface with the PAI and Protolib libraries which support sockets and timers. Such interactions are not one way since the PAI interfaces provide the capability of attaching listeners to sockets and timers. Therefore AgentJ provides a complex array of JNI callback mechanisms to route events to the appropriate C++ objects. Since the JVM context cannot be sustained for any given node during simulation, the C++ agent pointers are essential to maintain that state.

The AgentJ nodes interface through the Java PAI libraries by using a JNI binding to the C++ PAI and underlying Protolib toolkits. An AgentJ node interacts with other AgentJ nodes by binding the standard Java interfaces to the PAI socket and timing classes within the JNI implementation, as indicated in Figure 4. This integration allows AgentJ nodes to issue commands for sending data or scheduling timers in NS-2.

For this implementation, a JNI interface is provided between the Java PAI interface and the corresponding C++ PAI interface that contains the necessary underlying functionality. Within this implementation, the node's *ID* is used along with the JNI binding to the PAI interface in order to re-associate the Java object with its C++ agent when we return back to the NS-2 context. In effect, what we are doing here is creating a JVM from C++, and using the JNI to re-enter C++.

Since the JNI C++ classes in the PAI and Protolib have no context, we cannot use static methods to obtain pointers to parts of the core NS-2 system. We circumvent this by transporting the pointers manually through the JVM so we can re-establish our context when we are back in the C++ world within NS-2. A Java application agent needs to be able to reference its corresponding C++ NS-2 agent so it can invoke calls to core NS-2 functions in an appropriate way.

B. Creating and Using AgentJ Nodes

Using AgentJ, each C++ NS-2 agent can (optionally) attach a Java agent. A Java NS-2 agent is a Java program that can be accessed through the standard AgentJ interface, which allows it to receive commands in the same ways as C++ NS-2 agents do. Thus, specifically, an AgentJ NS-2 agent is a Java object that implements the ***AgentJObject*** Java interface.

AgentJ nodes can use any 3rd party Java application in order to implement the behavior they require. In networked scenarios, where one application communicates with other instances of itself running on remote locations (represented as other agents (instances) within the AgentJ simulation), it would need to be able to use the AgentJ implementations of the Java network classes (described in Section V) in order to access the standard communication protocols and to schedule events and discover and communicate with other AgentJ nodes. Complete Java simulations can be built up using these simple primitives.

Java AgentJ objects that want to attach themselves to an NS-2 node must implement a simple command interface in order to receive commands from the NS-2 TCL simulation script (see Figure 5). Within the context of a Java application, this provide the gateway into scheduling the various parts of the Java simulation, which is initially triggered by the TCL script.

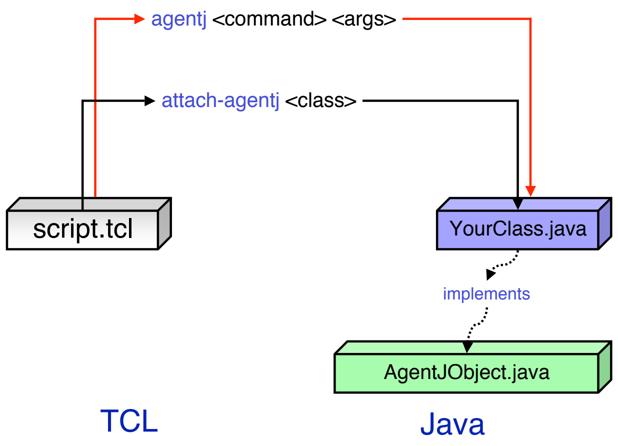


Fig. 5. Java agents can be attached to a node in an NS-2 simulation by specifying their classname with the `attach-agentj` TCL command. Commands and arguments can be passed to those Java agents via the `agentj` TCL command.

Figure 5 illustrates how a Java application can be attached to an NS-2 node by specifying its classname as a parameter to the `attach-agentj` TCL command. User commands are sent to this Java application agent via the `agentj` TCL command.

The relationship between an NS-2 TCL script and its Java agents is very similar to the relationship between an NS-2 TCL script and its C++ agents. The Java interface implements the same custom of sending simple text commands through a command method interface (see Figure 6. Briefly, commands passed to the TCL `agentj` command are caught within the AgentJ C++ implementation, which overrides the Protolib default implementation of command (in NsProtoAgent). The C++ AgentJ class then locates the appropriate Java object, through a Hashtable lookup and invokes a method defined in `AgentJObject` called `command`, which each AgentJ object must implement. This results in the TCL string being passed to the appropriate Java object. Using the same command-style interaction satisfies some essential constraints:

- **Flexibility:** it will keep the flexibility of being able to use NS-2 agents in any way a programmer sees fit - the Java extensions are optional and any agent extending the *JavaAgent* can choose to use this functionality. However, the core C++ agent code can be programmed to incorporate other functionality needed beyond the scope of Java.
- **Simplicity:** the scalability issues and framework for interacting with 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 NS-2 agent and any attached Java class through the same familiar interface as NS-2 programmers currently interface between the TCL scripts and C++ agents.

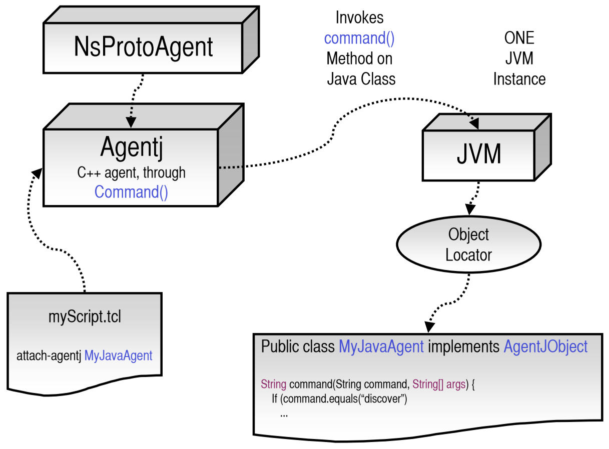
V. NS-2 Socket Implementation

AgentJ re-implements the various *java.net* classes to provide the socket-level functionality within NS-2. We also use our own class loader to rewrite the Java bytecode upon execution [16] to replace the *java.net.** occurrences with our equivalents, so that the AgentJ versions are used instead of original Java implementations. For a Java programmer, this means that little, or no modification is needed in order to integrate their code into NS-2.

We took this rather ad-hoc approach because of the limitations of the *java.net* package. The Java *Datagram* and *Socket* implementations are designed with the flexibility to plug in different socket implementations, using the *setDatagramSocketImplFactory* and *setSocketImplFactory* methods, respectively. This would have been the chosen route for the AgentJ implementation but this is impossible due to its reliance on the *InetAddress* classes. It is possible to the *SocketAddress* interface to circumvent most of these problems but even the packet implementations contain references to *InetAddress* and *InetAddress* cannot be subclassed or re-implemented out of its *java.net* context. *InetAddress* cannot handle different formats for addressing (such as NS-2), and therefore the *java.net* factory socket mechanism is only applicable to alternate implementations that use the Internet addressing scheme.

Further, within AgentJ, we need to provide a different implementation of *getLocalHost*, which is also implemented in *InetAddress*. In the conventional case, this returns the IP address of the machine running the Java application but in the case of AgentJ, this needs to return the NS-2 address of the node that is running this instance of the simulations. This is because AgentJ simulates a number of *nodes* of the distributed simulation on one machine.

Our current implementation addresses these problems by dynamically switching socket factories at the bytecode level. Although, this approach is not ideal, it does provide the Java programmer with mechanisms that allow him/her to simulate



```
attach-agentj MyJavaAgent
```

```
Public class MyJavaAgent implements AgentJObject

String command(String command, String[] args) {
   If (command.equals("discover")
      ...
```

Fig. 6. The user's view of the interaction between the NS-2 agent/script and the Java class associated with that NS-2 node.

their distributed applications with little code modifications. In practice however, modifications are often needed because most Java applications contain threads, which cannot be simulated currently within NS-2. We are currently investigating possibilities in order to provide AgentJ Java thread implementations for certain cases.

For example, within Java, it is not possible to attach a listener to a *DatagramSocket* to notify when data arrives. To accomplish this, a Java programmer would either block on the *receive* function or create a thread to monitor this and notify when data arrives. Neither approach is workable in NS-2: if you block on a method then control will not be returned to the NS-2 event dispatcher, data will not be processed and NS-2 will simply halt at that method; and if you spin off a thread then the results are non deterministic, e.g. NS-2 could finish before the thread returns.

VI. Conclusion

In this paper, the AgentJ framework was presented, which provides a Java interpretation capability to the NS-2 network simulator. The design of AgentJ adopts the standard NS-2 mechanisms for interacting between TCL scripts and C++ objects - thereby making it NS-2 user friendly. AgentJ supports complex interactions with Java objects and has implemented the dynamic creation of Java virtual machines in a manner that scales well to large network simulations. AgentJ also incorporates JNI interactions for mapping between C++ and Java objects, and for creating callbacks between numerous NS-2 agents that can be created within a simulation. AgentJ is on-going work that is in early *alpha* release. More code examples and documentation can be found on its Web site [4].

References

[1] "The Ns2 Simulator." See web site at: `http://www.isi.edu/nsnam/ns/`.
[2] I. Wang, "P2PS (Peer-to-Peer Simplified)," in *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pp. 54–59, Louisiana State University, February 2005.
[3] "Project JXTA," July 2005. `http://www.jxta.org`.
[4] "The Protean Research Group, SRSS Project, Naval Research Lab." See web site at: `http://cs.itd.nrl.navy.mil/5522/`.
[5] T. J, "Overlay Networks," *Computer Networks*, vol. 3 (2-3), pp. 115–116, 2001.
[6] "Gnutella." `http://www.gnutella.com/`.
[7] "LimeWire."
[8] "Kazaa."
[9] "Skype: Voice and Video Calling." See web site at: `http://www.skype.com/`.
[10] "Mobile Ad-hoc Networks (MANET)." See web site at: `http://www.ietf.org/html.charters/manet-charter.html`.
[11] "The Java Network Simulator." See web site at: `http://jns.sourceforge.net/`.
[12] "Javis: A Java NAM Visualization Tool." See web site at: `http://warriors.eecs.umich.edu/viz_tools/nam.html`.
[13] "DRCL J-Sim." See web site at: `http://www.j-sim.org`.
[14] "The Protolib Toolkit." See web site at: `http://proteantools.pf.itd.nrl.navy.mil/`.
[15] "The Java Native Interface Trail." See web site at: `http://java.sun.com/docs/books/tutorial/native1.1/`.
[16] A. Rudys and D. S. Wallach, "Enforcing Java Run-Time Properties Using Bytecode Rewriting," in *International Symposium on Software Security (Tokyo, Japan)*, November 2002.