JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 18, 981-997 (2002)

Neko: A Single Environment to Simulate and Prototype Distributed Algorithms*

PÉTER URBÁN, XAVIER DÉFAGO+ AND ANDRÉ SCHIPER
School of Computer and Communication Sciences
Swiss Federal Institute of Technology in Lausanne
EPFL, CH-1015 Lausanne, Switzerland
E-mail: peter.urban@epfl.ch, andre.schiper@epfl.ch
+Graduate School of Knowledge Science
Japan Advanced Institute of Science and Technology
Tatsunokuchi, Ishikawa 923-1292, Japan
E-mail: defago@jaist.ac.jp

Designing, tuning, and analyzing the performance of distributed algorithms and protocols are complex tasks. A major factor that contributes to this complexity is the fact that there is no single environment to support all phases of the development of a distributed algorithm. This paper presents *Neko*, an easy-to-use Java platform that provides a uniform and extensible environment for various phases of algorithm design and performance evaluation: prototyping, tuning, simulation, deployment, etc.

Keywords: simulation, prototyping, distributed algorithms, message passing, middleware, Java, protocol layers

1. INTRODUCTION

Designing, tuning, and analyzing the performance of distributed algorithms and protocols are complex tasks. Because of the performance requirements and the timing constraints of modern systems, performance engineering is an important activity in the construction of complex systems. Distributed systems are no exception, and the constant need for better performance is a strong incentive for proper performance engineering and algorithm tuning.

Performance engineering is based on a combination of three basic approaches to evaluating the performance of algorithms: (1) the *analytical approach* computes the performance of an algorithm based on a parameterized model of the execution environment; (2) *simulation* runs the algorithm in a simulated execution environment, usually based on a stochastic model; and (3) *measurement* runs the algorithm in a real environment. These three approaches have their respective advantages and limitations. Therefore, in order to increase the credibility and the accuracy of performance analysis, it is considered good practice to compare results obtained using at least two different approaches.

Received August 27, 2001; accepted April 15, 2002.
Communicated by Jang-Ping Sheu, Mokoto Takizawa and Myongsoon Park.
* Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel. A preliminary version of this paper appeared in [1].

In spite of its importance, performance engineering often does not receive the attention that it deserves. One reason stems from the fact that one usually has to develop one implementation of an algorithm for measurement, and a different implementation (possibly in a different language) for simulations. In this paper, we propose a solution to this last problem. We present *Neko* [2], a simple communication platform that allows us to both simulate a distributed algorithm and execute it on a real network, using the *same implementation* for the algorithm. Using Neko, thus, ultimately results in shorter development time for a given algorithm. Beside this main application, Neko is also a convenient implementation platform that does not incur a major overhead in communications. Neko is written in Java and is, thus, highly portable. It has been deliberately kept simple, extensible and easy to use.

The rest of the paper is structured as follows. Section 2 will describe the most important features of Neko. We will illustrate the simplicity of using Neko throughout this section. Section 3 will present the various types of real and simulated networks that Neko currently supports. Section 4 will describe applications developed with Neko. Section 5 will discusse other work that is related to Neko. Finally, section 6 will conclude the paper.

2. NEKO FEATURE TOUR

We will next give an overview of Neko. We will first present the architecture of the platform and the most important components seen by application programmers. We will then illustrate the use of Neko with a simple application. We will also show how to start up and configure the example, to run it either as a simulation or on a real network. Finally, we will discuss differences between simulations and distributed executions.

2.1 Architecture

As shown in Fig. 1, the architecture of Neko consists of two main parts: *application* and *networks* (NekoProcess will be explained later).

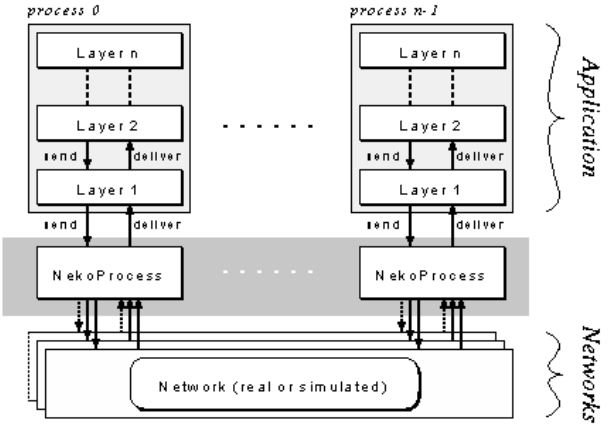

Fig. 1. Architecture of Neko.

At the level of the application, a collection of *processes* (numbered from 0 to $n - 1$) communicate using a simple message passing interface: a sender process pushes its message onto the network with the (asynchronous) primitive send, and the network then pushes that message onto the receiving process with deliver. Processes are programmed as multi-layered programs.

In Neko, the communication platform is not a black box: the communication infrastructure can be controlled in several ways. First, a network can be instantiated from a collection of predefined networks, such as a real network using TCP/IP or simulated Ethernet. Second, Neko can manage several networks in parallel. Third, networks that extend the current framework can be easily programmed and added.

We will next present some important aspects of the architecture that are relevant to Neko applications. Details related to the networks will be given in section 3.

Application layers. Neko applications are usually constructed as a *hierarchy of layers*. Messages to be sent are passed down the hierarchy using the method send, and messages delivered are passed up the hierarchy using the method deliver (Fig. 1). Layers are either *active* or *passive*. Passive layers (Fig. 2) do not have their own thread of control. Messages are pushed upon passive layers; i.e., the layer below calls their deliver method. Active layers (Fig. 3), derived from the class ActiveLayer, have their own thread of control. They actively pull messages from the layer below, using receive (they have an associated FIFO message queue supplied by deliver and read by receive). A call to receive blocks until a message is available. One can also specify a timeout, and a timeout of zero corresponds to a non-blocking call to receive.

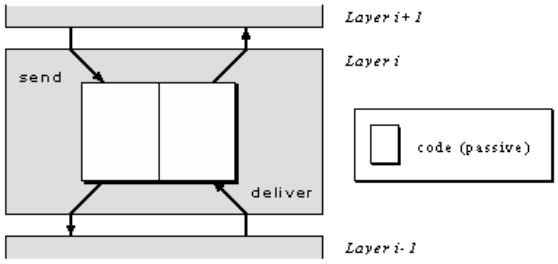

Fig. 2. Details of a passive layer.

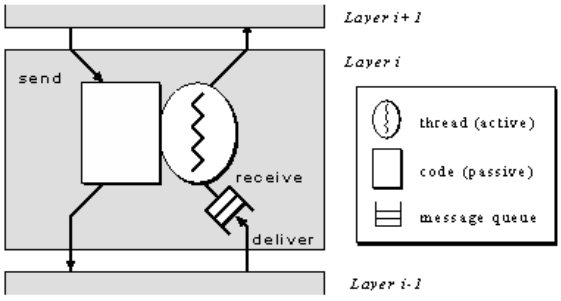

Fig. 3. Details of an active layer.

Active layers might interact with the layers below using deliver, just like passive layers do (Fig. 2). In order to do this, they have to bypass the FIFO message queue shown in Fig. 3 by providing their own deliver method.

Developers are not obliged to structure their applications as a hierarchy of layers. Layers can be combined in other ways: Fig. 4 shows a layer that multiplexes messages coming from several layers into one channel (and demultiplexes in the opposite direction), based on the message type. Layers may also interact by calling user-defined methods on each other; i.e., they are not restricted to send and deliver/receive. In general, developers may use Java objects of any type, not just layers, arranged and interacting in an arbitrary fashion.

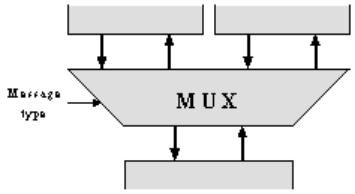

Fig. 4. Sample layer that multiplexes messages coming from upper layers.

NekoProcess Each process of the distributed application has an associated NekoProcess object, placed between the layers of the application and the network (Fig. 1). The NekoProcess plays several important roles:

1. It holds some process wide information; e.g., the address of the process. All the layers of the process have access to the NekoProcess. A typical use is Single Program Multiple Data (SPMD) programming: the same program is running on several processes, and it branches on the address of the process on which it is running. This address is obtained from the NekoProcess.
2. It implements some generally useful services, such as logging messages sent and received by the process.
3. If the application uses several networks in parallel (e.g., because it communicates over two different physical networks, or uses two different protocols over the same physical network), the NekoProcess dispatches (and collects) messages to (from) the appropriate network.

NekoMessage All communication primitives (send, deliver and receive) transmit instances of NekoMessages. A message can be either a unicast or a multicast message. Every message is composed of a content part that consists of any Java object and a header with the following information:

Addressing (source, destinations) The addressing information consists of the address of the sender process and the address of the destination process(es). Addresses are small integers; the numbering of processes starts from 0. This gives a very simple addressing scheme with no hierarchy.

Network When Neko manages several networks in parallel (see Fig. 1), each message carries the identification of the network that should be used for transmission. This can be specified when the message is sent.

Message type Each message has a user-defined type field (integer). It can be used to distinguish messages belonging to different protocols.

2.2 Sample Application: Farm of Processors

In this section, we will illustrate the application layers using an example. The example will be explained in detail in order to highlight how easily one can develop distributed applications with Neko. More complex applications are described in section 4.

Our example is as follows: a complex task is divided into sub-tasks, and each sub-task is assigned to one machine out of a pool of machines. When a machine has finished a sub-task, it sends back the result and gets a new sub-task. We also have a fault tolerance requirement. As we use a large number of machines, it is most likely that a few of them will be down from time to time. We do not want to assign sub-tasks to these machines.

The implementation has two layers on every process, as shown in Fig. 5. We will next describe these layers.

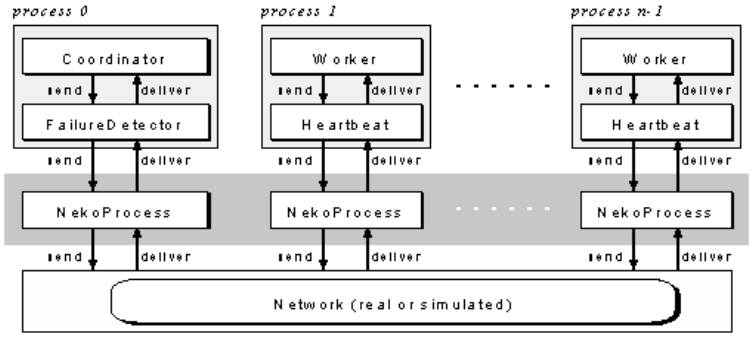

Fig. 5. Architecture of a sample Neko application: farm of processors.

Top layers. The Coordinator distributes the task and collects the result, and the Workers do the actual computation. The Worker code is shown in Fig. 6 (we only give code for the simplest layers due to space constraints). It is an active layer: active layers extend ActiveLayer, which extends NekoThread, which is used much as a Java thread is used. The thread executes its run method. This method uses receive to get a message from the network. The result is computed and sent to the layer below (stored in the sender variable).[1] After going through the protocol stacks, this message is delivered to the Coordinator.

The Worker layer can also be implemented as a passive layer, extending Layer; this implementation is shown in Fig. 7. Whenever a message is delivered to the layer, it computes some result and sends the message to the layer below (sender).

[1] The layer above is stored in the receiver variable.

```
public class Worker
   extends ActiveLayer
{
   public void run() {
      while (true) {
         //receive work from coordinator
         NekoMessage m = receive();
         //process work
         Object result = compute(m.getContent());
         //send back results
         sender.send(new NekoMessage(0, RESULT, result));
} } }
```

Fig. 6. Code example for an active layer.

```
public class Worker
   extends Layer
{
   public void deliver(NekoMessage m) {
      //process work received from coordinator
      Object result = compute(m.getContent());
      //send back results
      sender.send(new NekoMessage(0, RESULT, result));
} }
```

Fig. 7. Code example for passive layer.

Bottom layers. By describing the bottom layers, we intend to show here a code example and a few possible uses of the hierarchy of layers.

The Heartbeat and FailureDetector layers cooperate to implement failure detection. The basic idea is that Heartbeat layers send a *heartbeat* message every second, and the FailureDetector starts suspecting a process p if no heartbeat arrives from p in 2 seconds. When there is suspicion, the FailureDetector delivers a NekoMessage containing a notification to the Coordinator, which can then re-assign the sub-task in progress on the faulty process. This illustrates one possible use of the hierarchy of layers: notifying layers about asynchronous events (in this case, about the failure of a process).

In another use of the hierarchy of layers, lower layers can observe messages going to and coming from higher layers. We can exploit this to optimize the basic scheme for failure detection. Let replies from Workers also act as heartbeats, so that we can reduce the number of heartbeats. In other words, a Heartbeat only needs to send an (explicit) heartbeat if no reply has been sent for 1 second, and the Failure Detector only needs to suspect a process p if no reply *or* Heartbeat is received for 2 seconds. The code for a Heartbeat is shown in Fig. 8. It is an intermediate layer, with the methods send and deliver. The method deliver simply passes on messages; it uses the data member receiver that points to the layer on top (in this case, the Worker). The method send also passes on

```
public class Heartbeat
   extends ActiveLayer
{
   double deadline = clock() + PERIOD;

   public void send(NekoMessage m) {
      deadline = clock() + PERIOD;
      //PERIOD is 1 second
      sender.send(m);
      //sender is the layer below: the network
   }

   public void deliver(NekoMessage m) {
      receiver.deliver(m);
      //receiver is the Worker
   }

   public void run() {
      while (true) {
         sleep(deadline-clock());
         send(new NekoMessage(0, HEARTBEAT, null));
} } }
```

Fig. 8. Code example for an intermediate layer.

messages (and uses the data member sender that points to the NekoProcess below), but additionally, it sets the deadline variable, which indicates when the next heartbeat has to be sent. Heartbeat is an active layer; thus it extends ActiveLayer and has its own thread of control. Its run method takes care of sending a heartbeat message whenever deadline expires.[2]

2.3 Easy Startup and Configuration

Bootstrapping and configuring a distributed application are far from trivial tasks. In this section, we will explain what support Neko provides for this task.

Configuration. All aspects of a Neko application are configured by a single file. The name of this file is the only parameter passed to Neko at startup. Neko ensures that each process of the application has this information in the configuration file when the application starts. Possible configuration files for the example given in section 2.2 are shown in Fig. 9; distributed executions need the file shown in Fig. 9(a), and simulations need the file shown in Fig. 9(b). The entries will be explained in the remainder of this section.

Bootstrapping (Fig. 9, lines 1-2). Bootstrapping a Neko application is different for a simulation and a distributed execution; this is why some entries of the configuration files need to differ.

[2] A careful reader might notice that synchronized blocks are missing. Concurrent execution of Neko and the Heartbeat thread may indeed result in heartbeats sent more often than intended, but this will not compromise the integrity of the application.

```
1. process.num = 3
2. slave = host1.our.net,host2.our.net
3. network = lse.neko.networks.TCPNetwork
4. process.0.initializer = lse.neko.alg.CoordinatorInitializer
5. process.1.initializer = lse.neko.alg.WorkerInitializer
6. process.2.initializer = lse.neko.alg.WorkerInitializer
7. heartbeat.interval = 1000
```

(a) For a distributed execution.

```
1. process.num = 3
3. network = lse.neko.networks.MetricNetwork
4. process.0.initializer = lse.neko.alg.CoordinatorInitializer
5. process.1.initializer = lse.neko.alg.WorkerInitializer
6. process.2.initializer = lse.neko.alg.WorkerInitializer
7. heartbeat.interval = 1000
```

(b) For a simulation.

Fig. 9. Example of a Neko configuration file (line numbers are not part of the file).

Launching a simulation is simple: only one Java Virtual Machine (JVM) is needed along with the name of the configuration file, and Neko creates and initializes all processes. The number of processes is specified by the entry process.num.

For a distributed execution, one has to launch one JVM per process. The process launched last (called *master*) reads the configuration file and interprets the slave entry, which lists the addresses of all the other processes (called *slaves*). Then it builds a *control network*, including all the processes (by opening TCP connections to each slave), and distributes the configuration information.[3]

It is tedious to launch all the JVMs by hand for each execution of an application. For this reason, Neko provides *slave factories* to launch slaves automatically. The slave factories run as daemons on their hosts. When bootstrapping begins, the *master* contacts each slave factory and instructs it to create a new JVM, which will act as a slave. Also, there exists another way of bootstrapping that is useful in clusters where a command is provided to launch processes on multiple machines, like the mpirun script in an MPI environment [3, 4]. Here, all JVMs are launched at the same time and establish connections based on the information in the configuration file.

Initialization (Fig. 9, lines 3-7). The networks are initialized next (Fig. 9, line 3). The names of the classes implementing the networks are given by the network entry, which is, of course, different for simulations and distributed executions. The components of a real network in different JVMs usually need to exchange information upon initialization; the control network, built in the bootstrapping phase, is used for this purpose.

Then comes initialization of the application (Fig. 9, lines 4-6). Each process has an initializer class, given by the process./.initializer entry for process #*i*. The initializer

[3] The master-slave distinction only exists during bootstrapping. All processes are equal afterwards.

code of process #1 is shown in Fig. 10. It constructs the hierarchy of layers in a bottom-up manner by calling addLayer on the NekoProcess. The code has access to all the configuration information; it uses the (application specific) entry heartbeat.interval (Fig. 9, line 7) to configure the Heartbeat layer.

```
public class WorkerInitializer
   implements NekoProcessInitializer
{
   public void init(NekoProcess process, Configurations config) {
      Heartbeat heartbeat = new Heartbeat();
      process.addLayer(heartbeat);
      Worker algorithm = new Worker();
      process.addLayer(algorithm);
      heartbeat.setInterval(config.getInteger("heartbeat.interval") );
} }
```

Fig. 10. Code example for initializing an application.

Execution. Once initialization is finished, all NekoThreads are started, and the application begins to execute. The application has access to the entries of the configuration file.

Shutdown. Terminating a distributed application is also an issue worth mentioning. There is no general (i.e., application independent) solution for this issue. Neko provides a shutdown function that any process can call and which results in shutting down all processes. Processes may implement more complex termination algorithms that end with calling the shutdown function. The termination algorithm executed by the shutdown function exchanges messages on the control network (built during the bootstrapping phase).

2.4 Simulation and Distributed Executions

One of the main goals of Neko is to allow the same application to run (1) as a simulation and (2) on top of a real network. These two execution modes are fundamentally different in some respects. This section summarizes the (few) rules that need to be followed if the application is to be run both as a simulation and as a distributed application.

No global variables. The first difference is that all the processes of a simulation run on the same Java Virtual Machine (JVM),[4] whereas a real application uses one JVM for each process. For this reason, code written for both execution modes should not rely on any global (static) variables. Global variables have two uses in simulations: they either keep (1) information private for one process, or (2) information global for the whole application. In the first case, the information should be accessed using the NekoProcess object as a key. In the second case, there are two choices. Either the information should be distributed using the network, or (if constant and available at startup) it can appear in the configuration file (see section 2.3).

[4] Simulations are not distributed. They only simulate distributeion.

Threads. The other difference lies in the threading model. Real applications use the class java.lang.Thread, whereas thread based discrete event simulation packages have their own special purpose threads, which maintain simulation time and are scheduled according to this simulation time. The two threading models usually do not have the same interface.[5] Both have operations specific to their application area. For example, SimJava [5], one of the simulation packages used in Neko, defines channels between active objects as a convenient way to have threads interact. Even the overlapping part of the interfaces is different; e.g., Java threads are started explicitly with start, while SimJava threads are started implicitly when the simulation begins.

Neko hides these differences by introducing NekoThread, which encapsulates the common functionality useful for most applications. All thethreads of the application have to extend this class or use it with a Runnable object. NekoThreads behave like simplified Java threads: in particular, they support wait, notify and notifyAll as a mechanism to synchronize threads. The differences are as follows:

- Threads started during initialization of application only begin to execute when the whole Neko application is started. This simplifies initialization of the application to a great extent.
- Time is represented with variables of the type double in Neko (and the unit is one millisecond). Therefore sleep and wait with a timeout take double as an argument.

The class NekoThread and more generally, all classes that have the same interface but different implementations in the two execution modes are implemented using the Strategy pattern [6]. This makes adding other execution modes rather easy. As an example, Neko already supports two simulation engines, SimJava and a more lightweight simulation engine developed for Neko. One could imagine integrating further simulation packages.

It must be noted that no restrictions apply if the application is only to be run in *one* of the execution modes. Thus, distributed applications may use all the features of Java and forget about NekoThreads, and simulations may exploit all the features of the simulation package.

3. NETWORKS

Neko networks constitute the lower layers of the architecture (see Fig. 1). The programmer specifies the network in the configuration file. No change is needed in the application code, even if one changes from a simulated network to a real one or vice versa. In this section, we will present real and simulated networks.

3.1 Real Networks

Real networks are built on top of Java sockets (or other networking libraries). They use Java serialization to represent the contents of NekoMessages on the wire.

TCPNetwork is built on top of TCP/IP. It guarantees reliable message delivery. A

[5] The reason is that most Java simulation packages were inspired by existing C or C++ packages and do not follow the Java philosophy of threading.

TCP connection is established upon startup between each pair of processes. UDPNetwork is built on top of UDP/IP and provides unreliable message delivery, which is sufficient for sending heartbeats in the example given in section 2.2. MulticastNetwork can send UDP/IP multicast messages. PMNetwork uses the PM portable communication library [7], which bypasses the TCP/IP stack to take advantage of low-latency cluster interconnection networks, like Myrinet. Finally, EnsembleNetwork integrates the Ensemble group communication framework [8, 9] into Neko to perform reliable IP multicasts.

Neko is focused on constructing prototypes. Nevertheless, we performed some measurements to evaluate Neko's performance. We compared Neko's performance with the performance of Java sockets, using both TCP and UDP. According to [10], the performance of Java and that of C sockets is rather similar (within 5%) with the newest generation of Java Virtual Machines (JVMs); hence, our comparison gives an indication of Neko's performance versus C sockets. We used the same benchmarks as [10], from IBM's SockPerf socket micro-benchmark suite [11], version 1.2.[6] The benchmarks are as follows:

TCP_RR_1 A one-byte message (request) is sent using TCP to another machine, which echoes it back (response). The TCP connection is set up in advance. The result is reported as a throughput rate of transactions per second, which is the inverse of the round-trip time for request and response. With Neko, we use NekoMessages with null content; they still include 4 bytes of useful data (type) and constitute the shortest messages we can send.

UDP_RR_1 The same as TCP_RR_1, using UDP instead of TCP as a transport protocol.

TCP_STREAM_8K A continuous stream of 8 kbyte messages is sent to another machine, which continuously receives them. The reported result is the bulk throughput in kilobytes per second.

We used two PCs running Red Hat Linux 7.0 (kernel 2.2.19). The PCs had Pentium III 766 MHz processors and 128 MB of RAM, and were interconnected by a 100 Base-TX Ethernet. The Java Virtual Machine was Sun's JDK 1.3.1. The absolute results, as well as the relative performance of Neko versus Java sockets, are summarized in Table 1. They show that Neko's performance reached at least 52% of Java performance in all the tests, with TCP_STREAM_8K performance reaching 82%. Response times were affected more than throughput. The overhead was probably due to (1) serialization and deserialization of NekoMessages, (2) the fact that Java objects, including NekoMessages, are allocated on the heap and (3) the necessity of having a separate thread that maps from the send-receive communication mechanism of sockets to the send-deliver mechanism of Neko.

We will continue working on performance optimization of Neko. The most promising directions are (1) adapting standard Java serialization so that it performs well for short messages (it is optimized for dumping long streams of objects into a file) and (2) improving its buffering strategy.

[6] These experiments did not benchmark all aspects of communication. Nevertheless, they should give an indication of the overhead imposed by Neko on the native sockets interface. The experiment CRR_64_8K was not performed, as it has no equivalent in Neko.

Table 1. Performance comparison of Neko and Java sockets.

performance	Neko	Java	relative
TCP_RR_1 [1/s]	4743	8634	55%
UDP_RR_1 [1/s]	4799	9185	52%
TCP_STREAM_8K [kbyte/s]	6688	8174	82%

3.2 Simulated Networks

Currently, Neko can simulate simplified versions of Ethernet, FDDI and CSMA-DCR [12] networks. Complex phenomena, like collision in Ethernet networks, are not modeled, as they do not influence the network behavior significantly at low loads. The models are motivated and described in detail in [13, 14]. Other models can be plugged into Neko easily, due to the simplicity of the network interface.

A different kind of simulated network proved to be useful in debugging distributed algorithms. The network delivers a message after a random amount of time, given by an exponential distribution. This network, although not particularly realistic in terms of modeling actual performance, usually "exercises" the algorithm more than an actual implementation, where the network tends to behave in a more deterministic way.

4. APPLICATIONS

This section will present three applications developed with Neko.

Group communication. Neko was used to develop various algorithms in the context of fault tolerant group communication (see [15] for a good introduction to related algorithms). Examples include: failure detector components [16], a variety of consensus algorithms, reliable broadcast algorithms and total order multicast algorithms, as well as generic components that help with putting these components together. Active replication was implemented using these components. They form the basis of a future group communication toolkit.

Evaluating the cost of distributed algorithms using performance metrics. Both measurement and simulation intervene rather late in the design of a distributed algorithm: the actual implementation environment, or a model thereof, has to be available to evaluate the performance of the algorithm. Before this stage, simple environment independent *performance metrics* can help predict the performance.

Widely used metrics include the time complexity, which is roughly the number of communication steps taken by the algorithm, and message complexity, which is the number of messages generated by the algorithm [17, 18]. Neko can evaluate these metrics, as they are special cases of simulated networks. Neko was also helpful in evaluating the contention-aware metrics described in [19, 20], which improve on the time and message complexity metrics by incorporating contention on the network and the hosts. Neko was also used in an extensive evaluation of five representative total order broadcast algorithms [19, 20].

Robust replicated service. The FLP impossibility result [12] states that the problem of Consensus cannot be solved in asynchronous distributed systems if processes may crash. We ran experiments to determine the robustness of a replicated service, which can be affected by the FLP impossibility result [22]. The experimental setup consisted of an actively replicated server and several clients issuing requests to the server. For the experiments, components that generated requests and collected results were developed. For our purposes, the network and the participating machines were overloaded to their limits, which presented additional challenges.

5. RELATED WORK

Prototyping and simulation tools. The *x*-kernel and the corresponding simulation tool *x*-sim [23] constitute an object oriented C framework that has similarities with Neko. It is designed for building (lower level) protocol stacks. Efficient execution of the resulting protocols is a major goal. Instead, Neko's focus is easy prototyping of distributed protocols/applications. Neko is a much smaller framework that supports building the application as a hierarchy of layers, which resemble *x*-kernel protocol stacks, but in a simpler, more flexible way: (1) a layer is not restricted to adding/removing headers and fragmenting/reassembling messages, and (2) a thread-per-layer (rather than thread-per-message) approach is possible. Moreover, Neko benefits from the advantage of using Java rather than C: serialization, and deserialization of complex data and multithreading are supported. The NEST simulation testbed [24] also supports prototyping to some extent. The code used for simulation is rather similar to UNIX networking code (in C): normally, only a few system calls have to be changed.

Microprotocol frameworks. Microprotocol frameworks aim at providing more flexible ways for protocol composition than traditional ISO/OSI stacks can. In these frameworks, protocol layers are only allowed to interact in well specified ways (e.g., a typical restriction is that layers cannot share a state; they have to communicate using events that travel up or down the protocol stack). As a result, the code for protocols tends to be more maintainable than code built in a more ad hoc manner, and new protocols or protocols with a new set of properties can be composed easily. *x*-kernel, Ensemble [8], Coyote and Cactus [25, 26] and Appia [27] are examples of such systems, and these systems were used to construct group communication protocols, just like Neko. Neko also composes protocols out of layers, but as we have used it primarily for prototyping so far, we have not felt the need for the advanced architecture that microprotocol frameworks have. If the need arises, we plan to integrate some of the ideas behind these frameworks into Neko.

Simulators. There exist a variety of systems developed to simulate network protocols. Most of them concentrate on only one networking environment or protocol; a notable exception is NS-2 [28], where the goal is to integrate the efforts of the network simulation community. These tools usually focus on the network layer (and the layers below), with (often graphical) support for constructing topologies, detailed models of protocols and network components. Neko is focused on the application layer, rather than on support for constructing complex network models. We see the two directions as comple-

mentary: in order to obtain realistic simulations based on detailed network models, Neko will have to be integrated with a realistic network simulator. The simplicity of Neko's network interface eases this task.

Message passing libraries. Neko (when used for prototyping) can be seen as a simplified socket library, with support for frequently occurring tasks like sending data structures, startup and configuration. A variety of simplified versions of the BSD C sockets interface are available (e.g., [29]). However, they are at best as easy to use as the Java interface for sockets. Other message passing standards exist: MPI [3, 4] and PVM [30]. They focus on different aspects of programming than Neko: they are mostly used in high performance computing to implement parallel algorithms, and efficient implementation on massively parallel processors and clusters is crucial. The result is that their APIs are complex compared with Neko: they provide operations that are useful in parallel programming but are hardly used in distributed systems: e.g., scatter, gather or reduce. The APIs also tend to be complex because they are C/Fortran style, even in Java implementations like mpiJava [31], jmpi [32], JPVM [33] and jPVM [34].

6. CONCLUSIONS

In this paper, we have presented Neko, a simple Java communication platform that provides support for simulating and prototyping distributed algorithms. The same implementation of an algorithm can be used both for simulations and executions on a real network; this reduces the time needed for performance evaluation and, thus, the overall development time. Neko is also a convenient implementation platform which does not incur a major overhead on communications. Neko is written in Java and is thus highly portable. It has been deliberately kept simple, easy to use and extensible: e.g., more types of real or simulated networks can be added or integrated easily.

We plan to continue developing Neko. Our short term goals include improving the efficiency of the built-in simulation package (SimJava), implementing more components useful for group communication and integrating a transport layer with an efficient (IP multicast based) reliable multicast protocol. A long term goal is integration with an advanced network simulator (such as NS-2). Also, we plan to switch to XML based configuration files, to have a cleaner way of configuring parts of a Neko application (i.e., configuration entries with a scope).

The Neko source code is available at no charge at http://lsrwww.epfl.ch/neko [2], along with documentation. Given sufficient interest, we will set up an Open Source project around it.

ACKNOWLEDGMENTS

Many thanks to Pawel Wojciechowski and Sergio Mena for their useful comments on microprotocol frameworks.

REFERENCES

1. P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," in *Proceedings of the 15th International Conference on Information Networking* (*ICOIN*-15), 2001, pp. 503-511.
2. P. Urbán, *The Neko Web Pages*, École Polytechnique Fédérale de Lausanne, Switzerland, 2000, http://lsrwww.epfl.ch/neko.
3. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference, Volume 1, The MPI-1 Core*, MIT Press, Cambridge, MA, USA, 2nd ed., Vol. 2, 1998.
4. W. Groupp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI: The Complete Reference, Volume 2, The MPI-2 Extensions*, MIT Press, Cambridge, MA, USA, 2nd ed., Vol. 1, 1998.
5. F. Howell and R. McNab, "SimJava: a discrete event simulation package for Java with applications in computer systems modeling," in *Proceedings of First International Conference on Web-based Modelling and Simulation*, *Society for Computer Simulation*, 1998; http://www.dcs.ed.ac.uk/home/hase/simjava/index.html.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software, Professional Computing Series*, Addison-Wesley Publishing Company, Reading, MA, 1995.
7. H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato, "PM: An operating system coordinated high performance communication library," in *Proceedings of High-Performance Computing and Networking Conference* (*HPCN*'97 *Europe*), 1997; http://pdswww.rwcp.or.jp/db/paper-E/1997/hpcn97/tezuka/tezuka.html.
8. R. Van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using ensemble," *Software: Practice and Experience*, Vol. 28, 1998, pp. 963-979.
9. M. Hayden, "The Ensemble system," Technical Report TR98-1662, Department of Computer Science, Cornell University, 1998.
10. R. Dimpsey, R. Arora, and K. Kuiper, "Java server performance: A case study of building efficient, scalable JVMs," *IBM Systems Journal*, Vol. 39, 2000; http://www.research.ibm.com/journal/sj/391/dimpsey.html.
11. IBM Corporation, *SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance*, 2000; http:/www.alphaWorks.ibm.com/aw.nsf/techmain/sockperf.
12. J.-F. Hermant and G. L. Lann, "A protocol and correctness proofs for real-time high-performance broadcast networks," in *Proceedings of 18th International Conference on Distributed Computing Systems* (*ICDS*-18), 1998, pp. 360-369.
13. N. Sergent, *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*, PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998; http://lsrwww.epfl.ch/Publications/Byld/35.html.
14. K. Tindell, A. Burns, and A. J. Wellings, "Analysis of hard real-time communications," *Real-Time Systems*, Vol. 9, 1995, pp. 147-171.
15. V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in S. Mullender (ed.), *Distributed Systems*, Addison Wesley, 2nd ed., pp. 97-146, 1993.

16. T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, Vol. 43, 1996, pp. 225-267.
17. N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
18. A. Schiper, "Early consensus in an asynchronous system with a weak failure detector," *Distributed Computing*, Vol. 10, 1997, pp. 149-157.
19. P. Urbán, X. Défago, and A. Schiper, "Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms," in *Proceedings of 9th IEEE International Conference on Computer Communications and Networks* (*IC3N* 2000), 2000, pp. 582-589.
20. X. Défago, *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*, PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000, No. 2229; http://lsrwww.epfl.ch/Publications/Byld/248.html.
21. M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, Vol. 32, 1985, pp. 374-382.
22. P. Urbán, X. Défago, and A. Schiper, "Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be?" in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems* (*SRDS*), 2001, pp. 190-193.
23. N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp. 64-76; http://www.cs.arizona.edu/xkernel/.
24. A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A network simulation and prototyping testbed," *Communications of the ACM*, Vol. 33, 1990, pp. 63-74.
25. N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote: A system for constructing fine-grain configurable communication services," *ACM Transactions on Computer Systems*, Vol. 16, 1998, pp. 321-366.
26. R. Schilichting and M. Hiltunen, *The Cactus Project*, 1999; http://www.cs.arizona.edu/cactus/.
27. H. Miranda, A. Pinto, and L. Rodrigues, "Appia: A flexible protocol kernel supporting multiple coordinated channels," in *21st International Conference on Distributed Computing Systems* (*ICDCS*'01), 2001, pp. 707-710.
28. K. Fall and K. Varadhan ed., *The ns Manual*, 2000, http://www.isi.edu/nsnam/ns.
29. Ch. E. Campbell Jr. and T. McRoberts, *the Simple Sockets Library*, http://users.erols.com/astronaut/ssl/.
30. V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, Vol. 2, 1990, pp. 315-339.
31. M. Baker, B. Carpenter, G. Fox, and S. H. Koo, "mpiJava: An object-oriented Java interface to MPI," LNCS, Vol. 1586, 1999, pp. 748-762.
32. K. Dincer, "A ubiquitous message passing interface: jmpi," in *Proceedings of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, 1999, pp. 203-207.
33. A. Ferrari, "JPVM: network parallel computing in Java," *Concurrency: Practice and Experience*, Vol. 10, 1998, pp. 985-992.
34. A. Beguelin and V. Sunderam, "Tools for monitoring, debugging, and programming in PVM," in *Proceedings of 3rd European PVM Conference* (*EuroPVM*'96), Vol. 1156, LNCS, 1996, pp. 7-13.

Péter Urbán is a research and teaching assistant at the Distributed Systems Laboratory at the Swiss Federal Institute of Technology in Lausanne (EPFL). He received his M.Sc. degree in Computer Science from the Budapest University of Technology and Economics. From 1997 to 1998, he worked at the CERN European Laboratory for Particle Physics in Geneva, Switzerland. His research interests include distributed systems, middleware, fault tolerance and performance evaluation.

Xavier Défago is a research associate at the School of Knowledge Science of the Japan Advanced Institute of Science and Technology (JAIST). He received a Ph.D. in Computer Science in 2000, from the Swiss Federal Institute of Technology in Lausanne (EPFL; Switzerland). From 1995 to 1996, he also worked at the NEC C&C Research Labs in Kawazaki (Japan). His research interests include distributed algorithms, fault-tolerance, computational robotics, and Grid systems.

André Schiper has been professor of Computer Science at the EPFL (Federal Institute of Technology in Lausanne) since 1985, leading the Distributed Systems Laboratory. During the academic year 1992-93 he was on sabbatical leave at the University of Cornell, Ithaca (NY). His research interests are in the areas of fault-tolerant distributed systems, middleware, group communication, and recently mobile ad-hoc networks. He took part in the following European projects: ESPRIT BROADCAST (1992-1995, 1996-1999), ESPRIT R&D OpenDREAMS I and II (1996, 1997-1999), IST REMUNE (2001-2003). He is member of the IST Network of Excellence in Distributed and Dependable Computing Systems (CABERNET). From 2000 to 2002, he is the chair of the steering committee of the International Symposium on Distributed Computing (DISC).