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

Péter Urbán[†]
peter.urban@epfl.ch

Xavier Défago[‡]
defago@jaist.ac.jp

André Schiper[†]
andre.schiper@epfl.ch

[†] *Communication Systems Department*
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland

[‡] *Graduate School of Knowledge Science*
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan

Abstract

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 the various phases of algorithm design and performance evaluation: prototyping, tuning, simulation, deployment, etc.*

Keywords: *simulation, prototyping, distributed algorithms, message passing, middleware, Java.*

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 greater performance gives a strong incentive for proper performance engineering and algorithm tuning.

Performance engineering is based on a combination of three basic approaches to evaluate 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) *measurements* run the algorithm in a real environment. These three approaches have their respective advantages and limitations. So, in order to increase the credibility and the accuracy of performance analysis, it is considered good practice to compare the results obtained through at least two different approaches.

In spite of its importance, performance engineering often does not receive the attention that it deserves. Part of the difficulties stems from the fact that one usually has to develop one implementation of the algorithm for measurements, 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* [26], a simple communication platform that allows 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 lower development time for a given algorithm. Beside this main application, 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 was deliberately kept simple, extensible and easy to use.

The rest of the paper is structured as follows. Section 2 describes the most important features of Neko. We intend to illustrate the simplicity of using Neko throughout this section. Section 3 presents the various types of real and simulated networks that Neko currently supports. Section 4 describes applications developed with Neko. Section 5 discusses other work that relates to Neko. Finally, Section 6 concludes the

[*] Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel.

paper.

2 Neko Feature Tour

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

2.1 Architecture

As shown in Figure 1, the architecture of Neko consists of two main parts: *application* and *networks*.

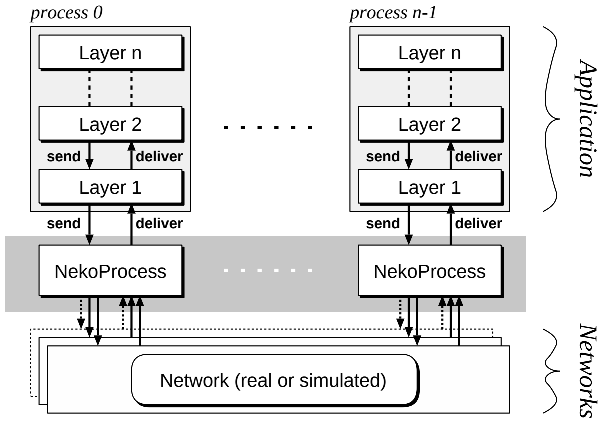

Figure 1. Architecture of Neko

At the level of the application, a collection of *processes* (process 0 to process $n-1$) 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 real TCP or simulated Ethernet. Second, Neko can manage several networks in parallel. Third, networks that extend the current framework can easily be programmed and added.

We now present some important aspects of the architecture that are relevant at the level of Neko applications. Details related to the networks are explained 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 (Figure 1). Layers are either *active* or *passive*. Passive layers (Figure 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 (Figure 3), derived from the class NekoThread, 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.) The 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.

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

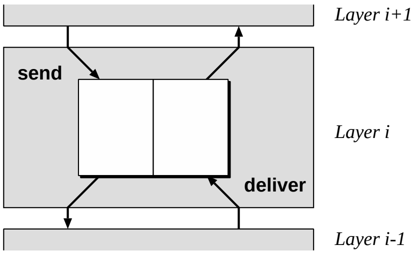

Figure 2. Details of a passive layer

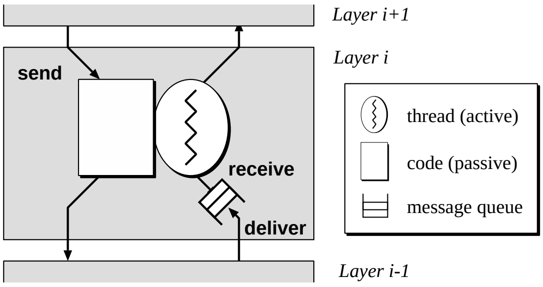

Figure 3. Details of an active layer

Developers are not obliged to structure their applications as a hierarchy of layers. Layers can be combined in other ways: Figure 4 shows a layer which

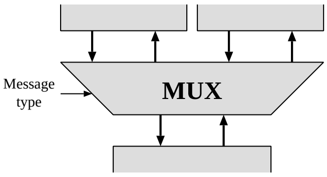

Figure 4. Sample layer that multiplexes messages coming from the upper layers

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.

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

1. It holds some process wide information; e.g., the address of the process and the local time. All layers of the process have access to the NekoProcess. A typical use is for 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.

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 (and from) the appropriate network.

NekoMessage All communication primitives (send, deliver and receive) transmit instances of NekoMessages. Sending a message can be either a unicast or a multicast. 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 Figure 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 illustrate the application layer using an example. The example is simple, but is explained in detail in order to highlight how easily one can develop distributed applications with Neko. Some more complex applications are described in Section 4.

Our example is the following: 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 are 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 Figure 5. We now describe these layers.

Top layers. The Coordinator distributes the task and collects the result, and the Workers do the actual computation. The Worker code is shown in Figure 6 (we only give code for the simplest layers due to space constraints). It is an active layer: active layers extend NekoThread, which is used similarly to a Java thread. The thread executes its run method. This method uses receive to get a message from the network. The result is computed and sent back to the Coordinator.

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 Failure Detector layers cooperate to implement failure detection. The basic scheme is that Heartbeat components send a *heartbeat* message every second, and the Failure Detector starts suspecting a process p if no

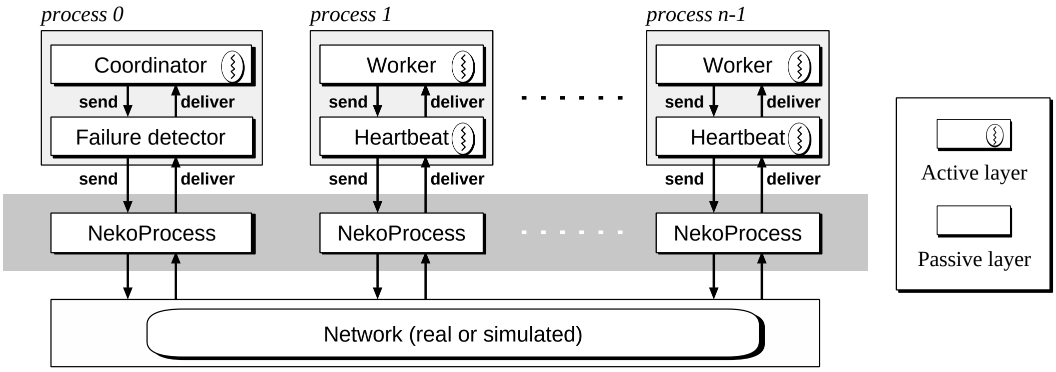

Figure 5. Architecture of a sample Neko application: farm of processors

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

Figure 6. Code example for an active layer.

heartbeat arrived from p in 2 seconds. Upon suspicion, the Failure Detector 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: for notifying layers about asynchronous events (in this case, about the failure of a process).

Another use of the hierarchy of layers is that 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 *nor* heartbeat is received for 2 seconds. The code for a Heartbeat is shown in Figure 7. It is an intermediate layer, with methods send and deliver.[1] The method deliver simply passes on messages; it uses the data member receiver which points to the layer on top (in this case, the Worker). The method send also passes on 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 NekoThread and has its own thread of control. Its run method takes care of sending a heartbeat message whenever the deadline expires.[2]

2.3 Easy Startup and Configuration

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

The startup (and other aspects) of Neko applications are controlled by a single configuration file. The name of the configuration file is the only parameter to be passed to Neko on startup. The configuration files of the example of Section 2.2 is shown in Figure 8; real executions need the file in Figure 8(a) and simulations the file in Figure 8(b) (most entries are the same).

Let n denote the total number of processes (specified by the entry process.num). Real executions are bootstrapped by one of the processes, called *master*. It reads the configuration file, and distributes the configuration information to all other processes, called

[1] These methods are part of FilterInterface that the Heartbeat class implements.

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

```
class Heartbeat
    extends NekoThread
    implements FilterInterface
{
    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());
            if (clock() >= deadline) {
                send(new NekoMessage
                        (0, HEARTBEAT, null));
} } } }
```

Figure 7. Code example for an intermediate layer.

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

(a) for a real execution

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

(b) for a simulation

Figure 8. Example of a Neko configuration file

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

Figure 9. Code example for initializing an application

slaves.[3] The addresses of slaves are specified in the slave entry. Simulations run in a single machine, so there is no bootstrapping problem and no slave entry.

The network is initialized next. The name of the class implementing the network is given by the network entry, which is, of course, different for simulations and real executions.

Then comes the initialization of the application. Each process has an initializer class, given by the process.*i*.initializer entry for process #*i*. The initializer code of process #1 is shown in Figure 9. It constructs the hierarchy of layers in a bottom-up manner by calling addFilter and registerReceiver on the NekoProcess. The code has access to all configuration information; it uses the (application specific) entry heartbeat.interval to configure the Heartbeat layer. Once all the initialization is finished, all NekoThreads are started and the application begins executing.

Terminating a distributed application is also an issue worth mentioning. There is no general (i.e., application independent) solution to this issue. Neko provides a shutdown function that any process can call and that results in shutting down all processes. Processes may implement more complex termination algorithms that end with calling the shutdown function.

2.4 Simulation and Distributed Applications

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. However, these two execution modes are fundamentally different in some respects. This section summarizes the (few) rules to follow 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 processes of a simulation run in the same Java Vir-

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

tual 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 to one process, or (2) information global to 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 is distributed using the network, or (if available at startup) it can appear in the configuration file (see Section 2.3).

Threads. The other issue is the threading model. Real applications use java.lang.Thread, and discrete event simulation packages have their 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. Both have operations specific to their application area. For example, SimJava [17], the simulation package 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 upon the start of the simulation.

Neko hides these differences by introducing NekoThread, which encapsulates the common functionality useful for most applications. All threads of the application have to extend this class. NekoThreads behave like simplified Java threads, except for the following:

- Threads started only begin execution when the whole Neko application is started. This simplifies the initialization of the application to a great extent.
- sleep takes a double as argument (and cannot be interrupted).

The classes NekoThread, NekoProcess and more generally, all classes that have the same interface but a different implementation in the two execution modes are implemented using the Strategy pattern [13]. This makes adding other execution modes rather easy; as an example, one could imagine integrating a simulation package other than SimJava into Neko.

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 features of Java and forget about NekoThreads, and simulations may exploit all features of the simulation package.

3 Networks

Neko networks constitute the transport layer of the architecture (see Figure 1). The programmer specifies the network in the configuration file. No change is needed to the application code, not even if one changes from a simulated network to a real one or vice versa. In this section, we present real and simulated networks.

3.1 Real Networks

Real networks are built on top of Java sockets; they receive the IP addresses and port numbers of participating processes upon the startup of Neko. They use Java serialization for representing NekoMessages on the wire.

TCPNetwork is built on top of TCP/IP. It guarantees reliable message delivery. A 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 uses like sending heartbeats in the example of Section 2.2.

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. Notice that according to [6] the performance of Java and C sockets are rather close (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 [6], from IBM's SockPerf socket micro-benchmark suite [19], version 1.2.[5] The description of the benchmarks 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.

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

[5] These experiments do 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.

performance	Neko	Java	relative
TCP_RR_1 [1/s]	3745	4223	89%
UDP_RR_1 [1/s]	1785	3340	53%
TCP_STREAM [kbyte/s]	6358	11017	58%

Table 1. Performance comparison of Neko and Java sockets

With Neko, we use NekoMessages with null content; they still include 4 byte of useful data (type) and constitute the shortest messages we can send.

UDP_RR_1 The same experiment repeated using UDP.

TCP_STREAM_8k A continuous stream of 8 kbyte messages is sent to another machine, which continuously receives them. The reported result is bulk throughput in kilobytes per second.

We used two Sun Ultra-60 stations running Solaris 2.6, connected to an Ethernet switch at 100 Mbit/s. The JVM used was Sun's JDK v1.2.2_05. The absolute results, as well as the relative performance of Neko versus Java sockets, are summarized in Table 1. They show that Neko performance reaches at least 50% of Java performance in all tests, with TCP_RR performance reaching 89%. The overhead is 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 the performance optimization of Neko.

It is worthwhile to note that the UDP_RR performance of Java is worse than its TCP_RR performance, contrary to our expectations. The reason is probably that Sun put more effort into optimizing TCP than into optimizing UDP.[6]

3.2 Simulated Networks

Neko uses the SimJava [17] discrete event simulation library. We chose SimJava because it is relatively simple, it is written entirely in Java and the source code is available. Other simulation libraries (e.g., [5]) could be integrated with Neko; Section 2.4 mentions the issues to consider at the integration.

Currently, Neko can simulate simplified versions of Ethernet, FDDI and CSMA-DCR [16] 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 [22, 25]. 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 usually "exercises" the algorithm more than an actual implementation, where the network tends to behave in a more deterministic way.

4 Applications

This section presents 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 into related algorithms). Examples include: failure detector components [3], two Consensus algorithms, one Reliable Broadcast algorithm, one Total Order Broadcast algorithm, as well as generic components that help plugging these components together. Active Replication was implemented using these components. They form the base 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 that stage, simple but environment independent *performance metrics* can help predicting the performance.

Widely used metrics include 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 [20, 21]. 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 [27, 4], which improve on the time and message complexity metrics by incorporating contention on the network and the hosts. Neko

[6] Even more strangely, UDP performance got *worse* from JDK 1.1.5 to JDK 1.2.2_05 and JDK 1.2.2_05 to JDK 1.3.0 beta according to our measurements.

was used for an extensive evaluation of five representative total order broadcast algorithms [4, 27].

Illustrating the FLP impossibility result. 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 illustrate this result intuitively [28]. The experimental setup consisted of an actively replicated server and several clients issuing requests to the server. For conducting the experiments, components that generated requests and collected results were developed. For our purposes, the network and the participating machines had to be 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 [18] 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. Neko instead aims at the construction of distributed *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: a layer is not restricted to adding/removing and splitting/reassembling information, and message contents can be any Java objects, and a thread-per-layer (rather than thread-per-message) approach is possible. Moreover, Java (used in Neko) is in general easier to use for development than C.

The NEST simulation testbed [8] 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.

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 [10], 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 complementary: in order to obtain realistic simulations 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., [9]). However, they are at best as easy to use as the Java interface to sockets. Other message passing standards exist: MPI [23, 14] and PVM [24]. 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 useful in parallel programming but hardly used in distributed systems: e.g., scatter, gather or reduce. The APIs tend to be complex also because they are C/Fortran style, even in Java implementations like mpiJava [1], jmpi [7], JPVM [11] and jPVM [2].

6 Conclusions

In this paper, we presented Neko, a simple Java communication platform that provides support for simulating and prototyping distributed algorithms. The same implementation of the 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 was deliberately kept simple, easy to use and extensible: e.g., some more types of real or simulated networks could be added or integrated easily.

We plan to continue developing Neko. The short term goals include improving the efficiency of the built-in simulation package (SimJava), implementing some 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.

The Neko source code is available freely at http://lsewww.epfl.ch/neko [26], along with some documentation. Given sufficient interest, we will set up an Open Source project around it.

References

[1] M. Baker, B. Carpenter, G. Fox, and S. H. Koo. mpi-Java: An object-oriented Java interface to MPI. *Lecture Notes in Computer Science*, 1586:748–, 1999.

[2] A. Beguelin and V. Sunderam. Tools for monitoring, debugging, and programming in PVM. In *Proc. 3rd European PVM Conf. (EuroPVM'96)*, volume 1156 of *Lecture Notes in Computer Science*, Munich, Germany, Oct. 1996.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] X. Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Aug. 2000. Number 2229. http://lsewww.epfl.ch/Publications/ById/248.html.

[5] Department of Computing Science, University of Newcastle upon Tyne. *JavaSim User's Guide*, 1999. http://javasim.ncl.ac.uk/.

[6] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1), 2000. http://www.research.ibm.com/journal/sj/391/dimpsey.html.

[7] K. Dincer. A ubiquitous message passing interface: jmpi. In *Proc. 13th Int'l Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing)*, San Juan, Puerto Rico, U.S.A., Apr. 1999.

[8] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon. NEST: A network simulation and prototyping testbed. *Comm. ACM*, 33(10):63–74, Oct. 1990.

[9] C. E. Campbell, Jr. and T. McRoberts. *The Simple Sockets Library*. http://users.erols.com/astronaut/ssl/.

[10] K. Fall and Kannan Varadhan, editors. *The ns Manual*, 2000. http://www.isi.edu/nsnam/ns.

[11] A. Ferrari. JPVM: network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11–13):985–992, Sept. 1998.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[13] 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.

[14] W. Gropp, 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, second edition, 1998. See also volume 1 [23].

[15] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. Addison Wesley, second edition, 1993.

[16] J.-F. Hermant and G. Le Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. In *Proc. 18th Int'l Conf. on Distributed Computing Systems (ICDCS-18)*, pages 360–369, Amsterdam, The Netherlands, May 1998.

[17] F. Howell and R. McNab. SimJava: a discrete event simulation package for Java with applications in computer systems modelling. In *Proc. First Int'l Conf. on Web-based Modelling and Simulation*, San Diego, CA, USA, Jan. 1998. Society for Computer Simulation. http://www.dcs.ed.ac.uk/home/hase/simjava/index.html.

[18] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. Software Engineering*, 17(1):64–76, Jan. 1991. http://www.cs.arizona.edu/xkernel/.

[19] 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.

[20] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[21] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[22] N. Sergent. *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1808. http://lsewww.epfl.ch/Publications/ById/35.html.

[23] 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, second edition, Sept. 1998. See also volume 2 [14].

[24] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, Dec. 1990.

[25] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, Sept. 1995.

[26] P. Urbán. *The Neko web pages*. École Polytechnique Fédérale de Lausanne, Switzerland, 2000. http://lsewww.epfl.ch/neko.

[27] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proc. 9th IEEE Int'l Conf. on Computer Communications and Networks (IC3N 2000)*, Oct. 2000.

[28] P. Urbán, X. Défago, and A. Schiper. The FLP impossibility result: A pragmatic illustration. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, 2000.