Quantitative Evaluation using Neko tool: NekoStat Extensions

Lorenzo Falai, Andrea Bondavalli, Felicita Di Giandomenico

DSI Technical Report
Firenze, November 2004

1 Introduction

The approaches for the evaluation of performance, dependability, performability of a distributed system or algorithm are essentially three: *analytic*, *simulative* and *experimental*. With analytic approach we assess interesting metrics using a parametric model of the execution environment; using a simulative approach, we simulate the evolution of the algorithm in a defined environment, usually based on a stochastic model. Finally with an experimental approach we execute the algorithm in a real environment, in concurrency with a monitor that collects data and obtains measurements. The use of at least two approaches is commonly considered a good rule to increase the confidence and accuracy of the obtained results.

Neko is a framework that permits to define and analyze distributed algorithms; the same Neko-based implementation of an algorithm can be used for simulations and for experiments on a real network. In this way, the development and testing duration is lower than with a traditional approach, in which usually these two implementations are implemented in different times and using different languages (for example, SIMULA for simulations and C++ for prototype implementation). Neko was created in the Distributed Systems Lab of EPFL Institute in Lausanne, by Peter Urban, Xavier Defago and Prof. Andre Schiper; Neko was implemented in Java, and it is thus highly portable. A complete description of Neko can be found in [14, 13].

In this report we describe NekoStat, an extension of standard Neko tool made by ourselves. This extension permits simpler and more powerful quantitative analysis of distributed algorithms, using simulative and experimental approaches. As for Neko, we develop NekoStat with an idea in mind: to obtain a tool usable in similar way for the two approaches to analysis.

The rest of the report is organized as follow. In Section 2 we describe the basic architecture and usage of Neko framework. Section 3 contains a short introduction about quantitative evaluation using Neko, and we introduce the package NekoStat. Finally, Section 4 contains an installation and developer guide for NekoStat.

2 Neko

The architecture of Neko can be subdivided in three components (see Fig. 1): *applications*, *NekoProcesses* and *networks*.

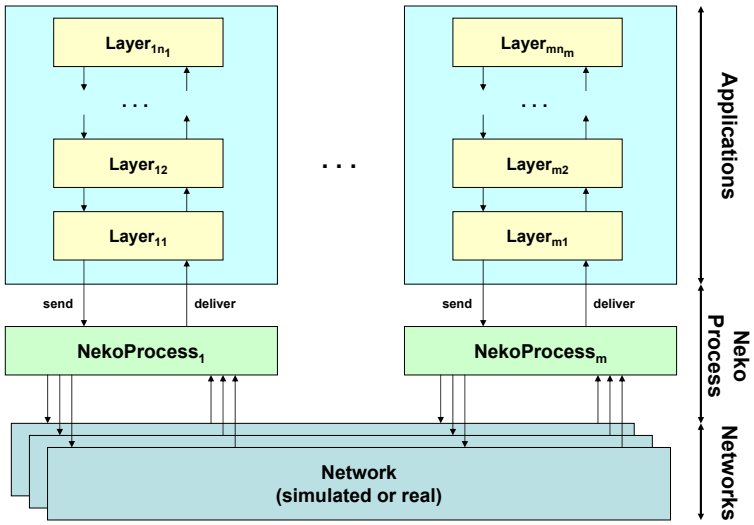

Figure 1: Neko architecture

A typical Neko-based distributed application is composed by a set of m processes, numbered $1, ..., m$, communicating by a message passing interface: a *sender process* inserts, with the asynchronous primitive send, a new message in the network, and the network deliveries the message to the *receiver process* with the primitive deliver. Applications are build with a structure based on multiple levels (called *Layers*).

Neko communication platform is a *white box*: the developer can use a network available on Neko or can define new network types; different networks can be used in parallel, and this permits to exchange different messages types within different networks (just to make an example, we can use a TCP connection for some message types and UDP datagram service for others).

2.1 Layers

Neko-based distributed applications are typically built on processes organized as hierarchy of levels, called layers. Layers can communicate using two predefined primitives for message passing: send transports a message from a level to level below, whereas deliver transports a message in the opposite direction.

There are two layer types: *actives* and *passives*.

Active layers have a control thread and a FIFO message queue that contains new available messages, inserted by bottom layer via deliver primitive. During

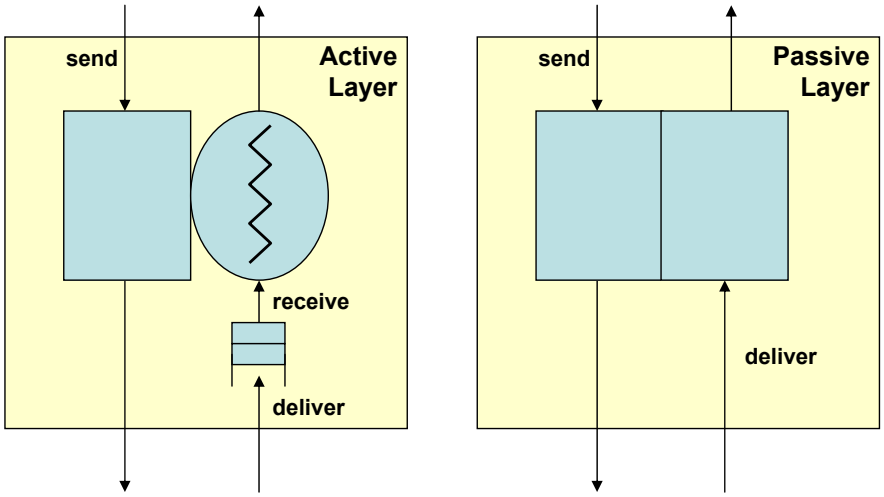

Figure 2: Active and passive layers

thread execution we can use receive primitive to take a new message from the queue (see Fig. 2). Default semantic of receive primitive is blocking: if there are not messages, control thread is stopped. It is possible to define also a maximum waiting time; a null time out corresponds to a non-blocking semantic.

Passive layers permit only messages transport from different layers, using send and deliver primitives (Fig. 2).

The usage of FIFO message queue for active layers is not mandatory: an active layer can provide a new (overloaded) deliver method. Also the layered architecture for the applications is not mandatory; layers can be combined in different ways, and can interact using different methods instead of the standard message passing primitives.

2.2 NekoProcesses

For every process that compose the distributed application exists an associated *NekoProcess*; this object stays between the lowest layer of the application and the networks. NekoProcess contains many informations useful for layers (for example: IP address of the node, Neko process identifier, ...) and it provides methods for common services (for example: it can log all sent and received messages, and it forwards messages to the appropriate network).

2.3 Networks

Neko *networks* are the lowest level of architecture of a Neko application. As above expressed in the presentation of Neko framework, an implementation of a distributed algorithm can be used whether on real execution (so using a real network) as on a simulated network, without changing a line of code. In Neko there are many predefined networks; developers can however implement new ones, both to use a proprietary network and to define a new kind of simulated one.

Neko communication platform is based on message passing; a message, (NekoMessage object) can be *unicast* or *multicast*. A message can contain any Java object, and it is characterized by a *header*, containing these informations:

(Source, Destinations): informations used to deliver the message to the destination (or to the destinations for a multicast message).

Network: information used to forward the message to the correct network.

Message type: every message has a type, defined by the developer. The message type is useful to distinguish messages from different protocols.

Neko networks are of two types: *real* and *simulated*. Real networks are built from Java sockets, or using external libraries for proprietary networks. Sending and receiving of a NekoMessage in real networks is based on *serialization*, operation that permits message representation during the message passing between processes.

Real predefined networks in Neko are: **TCPNetwork** (reliable connections between every pair of process), **UDPNetwork** (unreliable service of message delivery), **MulticastNetwork** (unreliable UDP-based multicast), **PMNetwork** (using PM library for low latency communication on cluster architecture - [12]), **EnsembleNetwork** (reliable multicast for IP, using Ensemble group communication framework - [5]).

Neko communication platform provides also several predefined simulated networks: **Ethernet**, **FDDI**, **CSMA-DCR**, and a network with an exponentially distributed message transmission delay (useful for debugging of distributed applications).

Integration of a new network type can be easily constructed: the developer has to define new model for the network and has to express it using a new NekoNetwork subclass (defining send and deliver methods).

2.4 Configuration, startup and shutdown of a Neko application

A Neko application can be configured with a configuration file, containing informations to prepare all processes.

In real executions there is an asymmetry between different processes: there is a *master process*, that coordinates the execution, and $m - 1$ *slave processes*. The master is the process that provide the configuration file to the slaves. For real execution we must have a master process and $m-1$ slave processes. Neko provides moreover *slave factories*, particular processes, always in execution, that waits new configuration file from a master process and executes new slave processes when needed. In real executions we thus have m processes running on m Java Virtual Machines, usually in execution on m different hosts, that communicate using the communication platform of the framework.

Simulation startup is simpler; we have m processes, in execution as different threads of one Java Virtual Machine.

Neko provides a predefined `shutdown` method to terminate a distributed application. The developer can however provided special shutdown functions.

2.5 How to make quantitative evaluation of a Neko application

Potentialities of Neko tool in the rapid prototyping of distributed algorithms are evident: the possibility to use simulated networks permits to analyze the algorithm in different conditions (variable transmission delays, different probability of message losses, network congestion, ...) and, after that, we can test the algorithm in real environments. Neko is thus very useful to test new algorithms, or to compare old ones, but it does not contain a direct support for *quantitative analysis* of the implemented algorithms. A quantitative analysis is usually necessary for the so-called *V&V* (verification and validation) process. The absence of a support of this kind carried us to study how we can evaluate, in a quantitative way, algorithms created using Neko framework; usable modalities can be summarized in three classes:

1. Realization of collection and analysis *tools, specific for a single distributed application*. Main drawback of this approach is the unnecessary rewriting of similar code for different applications.

2. *Direct analysis* of Neko log files. In Neko we can export all distributed events of a real or simulated execution on log files (using `java.util.logging`

framework); these files can contain all necessary informations to extract quantitative assessment for the interesting metrics. Main advantage of this approach is the simplicity in events processing, whereas main drawback is the impossibility in making on-line analysis for simulations: quantitative analysis is indeed realized only *after* the execution, and it is thus necessary off-line.

3. *Neko framework extension*, with appropriate tools directed to quantitative analysis; following this approach, we can define specific tools for simulation and real execution, and hide the differences creating a common interface for the tools. Main advantages are two: we can define this tool extension one time, and we can reuse it many, and this extension can easily permit on-line analysis (in concurrency with the execution) for simulations.

From the considerations expressed above, we decide to implement a Neko extension following the third approach. This extension was called NekoStat; in the next Section we describe NekoStat architecture, components and usage.

3 NekoStat

NekoStat was created as an extension of standard Neko tool, to permit a simpler and more powerful quantitative evaluation of distributed algorithms. One of the basic idea of Neko was to define a framework in which the developer can use one implementation of a distributed algorithm, whether in simulations or in real experiments. We wanted to keep this idea in NekoStat design: usage of provided tools is very similar in simulations and in real experiments.

NekoStat was realized to help in the collection and analysis of measures of a distributed algorithm. The measures are evaluated by *distributed events* that characterize the algorithm: a user defined *handler* transforms the events in quantities. We need only to follow these two rules to use NekoStat to measure a distributed algorithm:

1. introduce calls to `log(Event)` method of a special logger class (`StatLogger`), to signal the event occurrences;

2. implement a `StatHandler` that transforms distributed events of the algorithm in quantities.

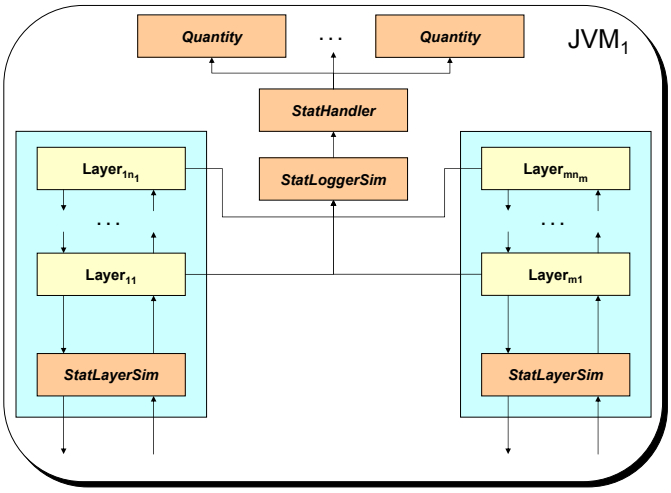

Figure 3: Typical NekoStat architecture for simulations

To analyze the functionality offered from NekoStat, in the next pages we describe the architecture of NekoStat and briefly all the components of the package.

In Fig. 3 and 4 are available the architectures of a typical analysis session with NekoStat, respectively for simulations and real executions. NekoStat tools can be subdivided in two sets: *mathematical tools*, that handles the quantities, and *analysis tools*, that collect and analyze distributed events. Mathematical tools are common between simulation and real executions, whereas analysis tools are internally different, also if with a common interface; differences in internal architecture are hidden to the developer, so as the analysis code can be reused without changes.

In the next Subsection we first describe mathematical tools of NekoStat, after that analysis tools, and finally we examine a simple example of NekoStat usage.

3.1 Mathematical tools for statistical analysis of quantities

Support tools for handling quantities was obtained as expansions of the *Colt* mathematical library for Java ([3]). Colt library was developed at Cern (Ginevre), with the purpose to offer a powerful and fast mathematical library for Java (in place of default tools available in Math package).

The classes defined for statistical analysis are composed of a *values container* and *methods* to obtain statistical parameters. The common interface available in all classes for statistical analysis of a quantity is as follow:

- one add(double) method, to insert a new double value in the quantity;

- methods to obtain statistical parameters of the quantity, as size(), mean(), getMedian(), standardDeviation(), min() and max(). Complete list of the information obtainable from a quantity can be found in Colt library documentation ([2]).

We built three different classes to handle quantities; differences are in obtainable informations and occupied memory:

- **QuantityOnlyStat**: this class does not maintain a complete list of values, and so we can obtain only main statistical parameters, as estimated mean, variance, standard deviation, but for example we cannot eliminate maximum and minimum outliers and we cannot make covariance analysis with other quantities. Usable methods are those defined for hep.aida.StaticBin1D (see [2])
- **QuantityDataOnFile**: it is a QuantityOnlyStat, with automatic export of all observed values on file.
- **Quantity**: this is most powerful class for quantities; usable methods are those defined for hep.aida.DynamicBin1d (see [2]). This class maintain a complete list of observed values, making possible to obtain more information about the quantity: for example, we can obtain covariance between two Quantity,

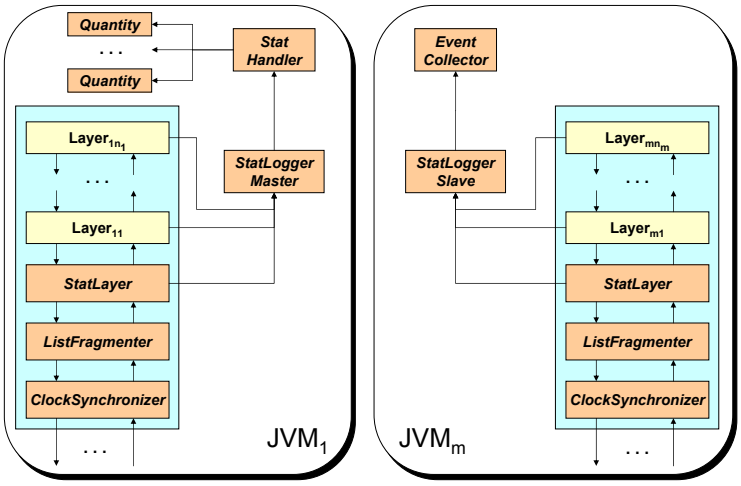

Figure 4: Typical NekoStat architecture for executions on real network

we can eliminate minimum and maximum outliers and final transient. The drawback is memory occupation for values container.

Common methods available for Quantity classes above defined are: initial transient elimination, export of main statistical parameters on file, definition and handling of stop conditions for simulation.

Stop condition is a boolean variable, whose value is true when observed values give enough confidence on the quantity. We defined two stop condition:

Stop after N values: after N collected measures of the quantity, stop condition become $true$;

Stop on confidence interval: stop condition become $true$ when confidence interval on mean, of level $(1-\alpha)$, is less than β percent of the mean. After n measures, let be $\widehat{x}$ estimated mean, S^2 estimated variance of the quantity; we thus decide when the data collected are enough using the expression:

$$t_{n-1,1-\frac{\alpha}{2}}\sqrt{\frac{S^2}{n}} \leq \widehat{x}\frac{\beta}{(1-\beta)}$$

See [7] to obtains further informations about the expression above.

We recall that use of a stop condition make sense only for on-line analysis in simulations; for experiments, quantities are evaluated after the complete distributed execution, and so we can use all collected events.

3.2 Analysis tools

This set of tool can be distinguished between tools common to simulations and real executions analysis, and tools that are used only for real executions.

3.2.1 Analysis tools common to simulations and experiments

We implemented the following analysis tools:

- StatLogger: it is the main NekoStat component, visible to the developer; it offers the method *handle(Event)*, that is used to inform NekoStat of a new event.

  In simulations, it receive the new event from a process thread, it calls the event handling routine defined for the analysis (handle(Event) method of

chosen StatHandler) and it evaluates if simulation can stop. In real execution, it collects the local events on a list and, when the distributed execution is stopped, it sends events list to the master process for successive analysis. StatLogger is based on Singleton pattern; in this way, all components belonging to a JVM see the same object.

- Event: it is the container for events informations. Every NekoStat event is characterized by: process identifier, time, description (a string), content (any Java Object). Time of an Event can be a real or a simulated time. Real time in Neko is measured in milliseconds from local NekoProcess start. In simulations all NekoProcesses see the same clock (clock of the simulator).

- StatHandler: it is the NekoStat component that must be provided by the developer; its role is the transformation of distributed events in quantities of interest for the analysis. A StatHandler must implement the handle(Event) method, in which we get a new event and we can evaluate and insert new measures in Quantity classes. For simulations we can also implement shouldStop method, that must return true when the collected data are enough (this method can use one or more stop condition defined on the analyzed quantities).

  NekoStat provides a standard implementation of a StatHandler, called LoggingStatHandler, that simply logs all handled events; it can be useful for a first, debugging, phase of the analysis process.

- StatInitializer: it is the component that prepare all the architecture visible in Fig. 3 e 4 for simulative and experimental analysis.

- StatLayer: it handles NekoStat control messages; its main role is in analysis termination phase.

  In simulations, when the StatLayer receive a message NS_STOP, it activate results exportation procedure and after that it stops the analysis. In real executions the master has an active role in the termination: StatLayer on the master sends a message NS_ASKEVENTS, and the StatLayer on slave processes reply with the collected events (NS_EVENTS); as we describe in the next Subsection, events are sent in a buffered way, to prevent situations in which master memory is not enough to contain all the history of the execution. The typical message exchange in a analysis termination phase for

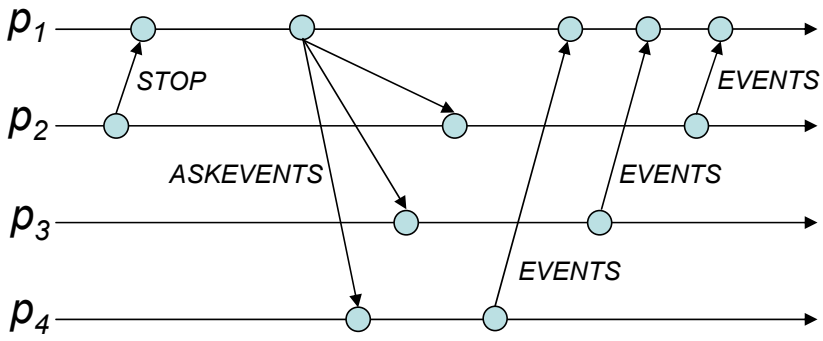

Figure 5: Analysis termination phase for real executions

a real execution is depicted in Fig. 5. Note that NekoStat must use a reliable connection for correct working of this termination protocol; in case of use of multiple networks, the developer can specify the network that must be used to exchange NekoStat control messages (with an entry of the configuration file, see Section 5).

3.2.2 Specific analysis tools for real executions

Some tools are used only for experimental analysis:

- EventCollector: it is the class used for local event collection during the experiment; it uses SparseArrayList to collect the events. It is a special kind of list, in which we prepare space only for a new value when reserved memory is exhausted. Using this approach, we diminished the overhead of monitoring process. Using a Vector, for example, can imply many milliseconds of local process stop when reserved memory is finished.
- ListFragmenter: it is used to send in a buffered way the local EventCollector, during the last phase of analysis. This layer sends to the master a subset of the events in the EventCollector; the events are sent in blocks to increase performance.

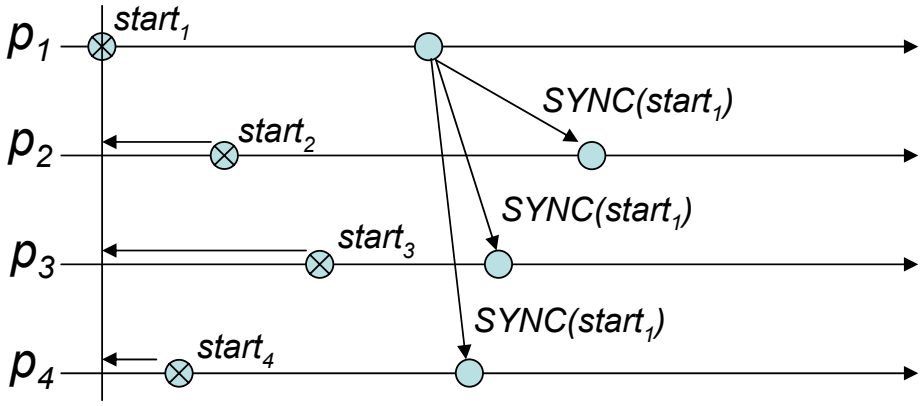

Figure 6: Logical clock synchronization provided by the ClockSynchronizer

- ClockSynchronizer: in real executions, it can be very important to have the clock of all processes synchronized. It permits to obtain *distributed duration* estimate. Having clocks synchronized permits to consider the clock as *global*; without a global clock we can only evaluate *round-trip durations*, intervals of time defined with start and end events occurring on the same site. For NekoStat we decided to use a synchronization with the real time *external* to the architecture, using the NTP protocol ([8, 11]); the ClockSynchronizer implements a simple master-slave algorithm to synchronize logical Neko clock of all processes with the logical clock of the master. At the beginning of the analysis, we execute this algorithm, and after that we have a unique global logical clock in all NekoProcesses, corresponding to a unique global clock (look at Fig. 6). This is the default solution: it is possible to insert in the NekoStat architecture a different clock synchronization algorithm, modifying the default ClockSynchronizer.

3.3 Analysis with NekoStat: a simple example

In Section 3 we described internal architecture of NekoStat; in this Section we describe more deeply analysis achievable with NekoStat. As support for successive explanation, Fig. 7 is space-time diagram corresponding to a portion of the execution of a reliable broadcast algorithm; note that the execution can be *simulated* or *real*. We suppose that the algorithm is implemented using Neko framework,

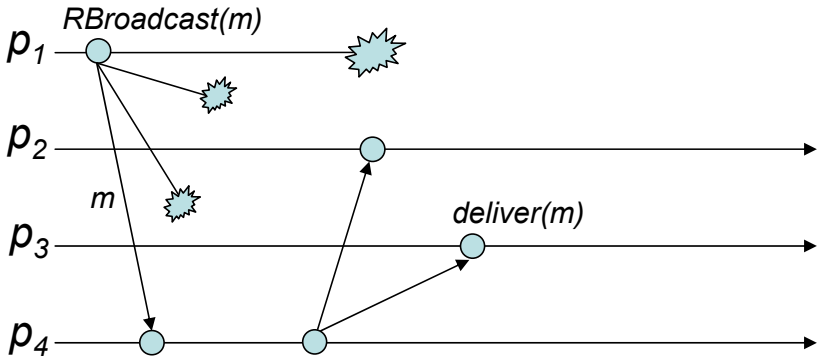

Figure 7: Space-time diagram of an example run of a reliable multicast algorithm (it can be a simulative or real execution diagram)

and that we want to evaluate **TimeToDeliver**, the interval of time between broadcast of message *m* and deliver of the message to the last process. We have thus to prepare an opportune StatHandler that transforms *RB* (ReliableBroadcast) and *deliver* events in the measure of interest; in this case it can make a simple difference between times of these events. In Fig. 8 and 9 we depicted components communication needed to obtain a new measure of the quantity **TimeToDeliver**; Fig. 8 is related to a simulative analysis whereas Fig. 9 is related to a real execution. As you can see also in the figures, in the first case we have an analysis *in concurrency* with the simulation, whereas in the second the analysis have to be made only after execution termination.

3.4 Simulative analysis

Analysis with NekoStat of a simulation of a distributed system starts with the preparation of StatLogger, StatHandler and StatLayer, created by StatInitializer. During the simulation, layers signal new events to the StatLogger; it calls immediately handle method of the StatHandler, in which we can obtain a new measure that can be inserted in the Quantity container. For the simulation we have thus *on-line* analysis. Collected informations about the quantities are exported on files when simulation finishes; the simulation can terminate or by explicit request of a layer, or when stop conditions are reached for all quantities.

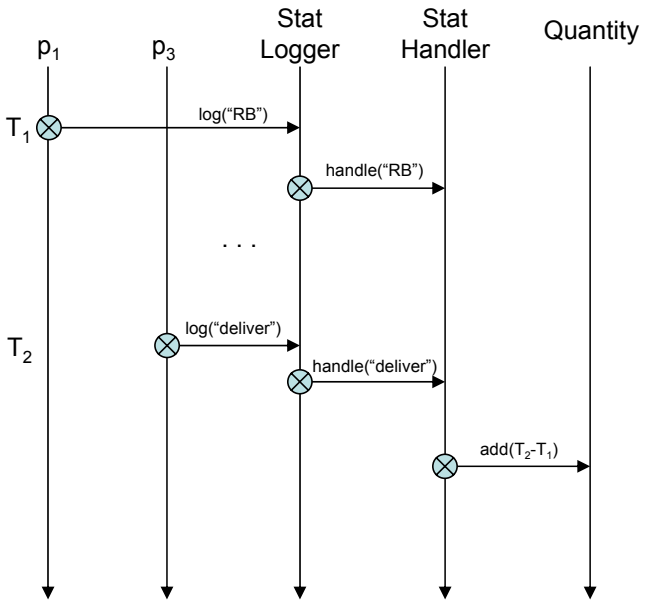

Figure 8: NekoStat analysis evolution for a simulation, depicted as a time diagram

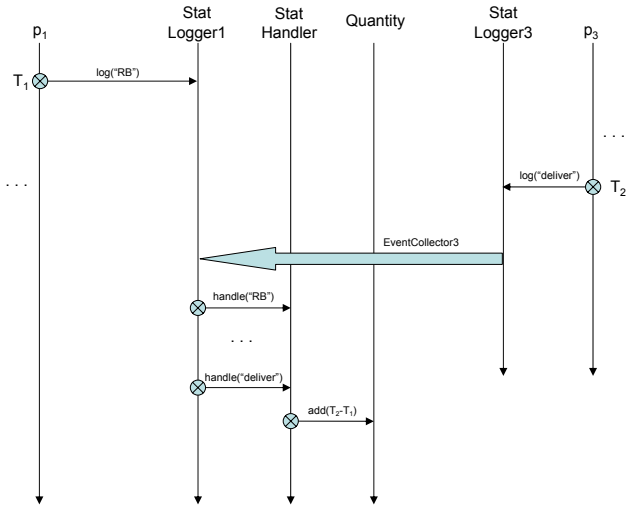

Figure 9: NekoStat analysis evolution for an experiment, depicted as a time diagram

3.5 Real distributed execution

Analysis with NekoStat of a distributed execution requires a first phase of initialization; the phase is slightly different between master and slaves. At the analysis beginning in the master the layers ClockSynchronizer, ListFragmenter and StatLayer, and the objects StatLogger and StatHandler are created. After that there is a clock synchronization phase, in which logical clocks of slave processes are synchronized with master logical clock. At the analysis beginning in the slaves the same layers as master initialization are created, also if with a different internal working logic, and StatLogger is created; StatHandler is instead active only in master process (analysis coordinator). At the end of this first initialization phase, Neko processes are created and started. During the execution, as you can see in Fig. 9, StatLogger collects local events on EventCollector. Any process can require analysis stop, sending a NS_STOP message to the master. At the reception of this message, StatLogger on master starts termination phase, in which it recreates complete history of the distributed execution to extract informations on the quantities. Analysis feasible for distributed executions is necessarily *off-line*: we do not use stop conditions and so the analysis must be interrupted by a process of the system.

4 NekoStat install and developer manual

In this Section we describe more deeply how to install NekoStat and we describe another simple example to understand better how to use NekoStat tools. NekoStat is an open-source project, and so you can modify available tools, or develop and integrate new ones; NekoStat license is the same of Neko one. Architecture of the library was built to be easily extensible. NekoStat will be integrated in the next official version of Neko tool; for now it is released as an add-on for standard Neko.

4.1 Installation

To install NekoStat you can follow these passes:

1. Download neko0.8.tar.gz, Neko 0.8, from [9], and decompress the file; the decompression creates neko directory, containing the tool;
2. Download nekostat0.8.tar.gz, NekoStat 0.8, from [10] and decompress this file in the directory containing neko as subdirectory. This decompression

installs NekoStat files in the Neko sources.

3. Now you can make a standard Neko installation, following the instruction from neko/README file.

NekoStat installation requires JAVA SDK with a version greater than 1.3. Since it is built with Java code, the tool can be usable on different architecture and operating system; anyway, some support executables and scripts are for now available only for Unix systems.

4.2 Tutorial: the package lse.neko.examples.stat

The package lse.neko.examples.stat contains an example of how to use NekoStat to make quantitative analysis; it is a simplified version of an analysis of the QoS of failure detectors that we described in [4].

The source files of the package are in neko/lse/neko/examples/stat directory, that contains:

FailureDetectorCTA.java ActiveLayer implementing NFD-S failure detection algorithm, described in [1];

FailureDetectorCTAInitializer.java it create the NekoProcess containing FailureDetectorCTA layer and NekoStat tools (see Fig. 10);

FailureDetectorCTAStatHandler.java it is the StatHandler that permits to evaluate T_M and T_{MR} metrics (see [1]) and one-way transmission delay of heartbeat messages;

HeartbeatCTA.java : ActiveLayer that implements the heartbeater;

```
public class FailureDetectorCTAInitializer
        implements NekoProcessInitializer
{

public void init(NekoProcess process, Configurations config) {
        new StatInitializer().init(process, config,
                        new FailureDetectorCTAStatHandler());
        // here the code for layers initialization
}
}
```

Figure 10: NekoStat initialization

HeartbeatCTAInitializer.java similar to FailureDetectorCTAStatHandler.java, but it creates the process containing HeartbeatCTA;

network.config, simulation.config configuration files, respectively for execution on real network (using UDP for heartbeat sending) and for simulations (using a simulated network with exponentially distributed transmission delay).

FailureDetectorCTAStatHandler is the StatHandler to make quantitative analysis. In Fig. 11 there is the portion of code that evaluates T_M metric. Main methods of this StatHandler are the following:

init(): called at analysis beginning, that corresponds to process start in case of

```
private QuantityDataOnFile tm;

public void init() {
        tm = new QuantityDataOnFile("tm");
        tm.setStopConditionInterval(0.95, 0.05);
}

public void handle(Event e) {
        if (e.getDescription().equals("startSuspect")) {
                // erroneous suspect start
                tmChrono = new Chronometer(e.getTime());
                return;
        }
        if (e.getDescription().equals("stopSuspect")) {
                // if actually 'suspected' –> 'trust'
                if (tmChrono != null) {
                        tmChrono.setStop(e.getTime());
                        tm.add(tmChrono.getValue());
                }
        return;
        }
}

public boolean shouldStop() {
        // method used in simulations
        // it permits to control the stop condition on the quantities
        return (tm.isStopReached());
}

public void writeResults() {
        // results exported on file
        tm.export();
}
```

Figure 11: StatHandler example: FailureDetectorStatHandler

simulations, and to distributed execution stop in case of experiments on real network. This method contains quantities initialization and it may contain stop condition setup for the quantities; for example, in this case we set a stop condition on T_M quantity, based on the dimension of confidence interval of 95% level (we consider the data collected enough when this interval is smaller than 5% of the mean).

handle(Event): it makes the transformation of events in quantities. In this case, for example, we can measure a new T_M value calculating the interval between successive $startSuspect$ and $endSuspect$ events (note that we consider monitored process always alive).

shouldStop: it is called only in simulations, after the handling of a new event; here we check stop conditions on quantities, to decide if the simulation can terminate.

writeResults(): it is called when the analysis terminate; here we define the informations that we want export. For example, in this case we use export() method of the quantity to write main statistical parameters of T_M on a file: mean, standard deviation, maximum, minimum and confidence interval on mean.

For real executions of this application we choose UDP datagram sending for the heartbeat; UDP sending is unreliable, and so we have to instantiate also a secondary, reliable, network: we chose TCPNetwork. In Fig. 12 is reported the portion of network.config configuration file where we describe networks to use for experiments. In Fig. 13 we report the portion of code of FailureDetectorCTA layer that permits to obtain T_M measures (note the calls to handle() method of the StatLogger).

Note that NekoStat source code is widely commented; for detailed informations on NekoStat internal working, we refers to this code, contained in lse.neko.stat package.

```
# Used networks:
# UDP for heartbeats, TCP for NekoStat messages
network=lse.neko.networks.comm.UDPNetwork,lse.neko.networks.comm.TCPNetwork
stat.network.index = 1
```

Figure 12: Portion of configuration file with networks definition

```
public class FailureDetectorCTA
        extends ActiveLayer {

public void deliver(NekoMessage m) {
[...]
        if (timestamp > lastTimestamp) {
                lastTimestamp = timestamp;
                if (timestamp >= (pass - 1)) {
                        if (suspected) {
                        statLogger.log("stopSuspect", null);
                        suspected = false;
                }
        }
[...]
}

public void run() {
[...]
while (true) {
        deadline = initialClock + pass * eta + delta;
        sleep(deadline - clock());
        if (lastTimestamp < pass) {
                if (!suspected) {
                        suspected = true;
                        statLogger.log("startSuspect", null);
                }
        }
pass++;
[...]
}
```

Figure 13: Portion of code of FailureDetectorCTA with the events needed to evaluate T_M

5 Acknowledgments

For NekoStat realization we collaborate with one of Neko creators, Peter Urban. We like to thank him very much for his collaboration and for his advices. We thank also Prof. André Schiper for his useful comments on NekoStat design.

References

[1] W. Chen, S. Toueg, M. K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, Vol. 51(5), 2002.

[2] Colt Library API documentation.
http://dsd.lbl.gov/~hoschek/colt/api/index.html

[3] Colt Library website.
http://dsd.lbl.gov/~hoschek/colt/

[4] L. Falai, A. Bondavalli. Experimental Evaluation of the QoS of Failure Detectors on Wide Area Network, *to publish*.

[5] M. Hayden. The Ensemble system. *Technical Report TR98-1662*, Cornell University, 1998.

[6] $JAVA^{TM}$ 2 Platform, Standard Edition, ver. 1.4.2 - API Specification.
http://java.sun.com/j2se/1.4.2/docs/api/

[7] A. M. Law, W. D. Kelton. Simulation, Modeling and Analysis. *McGraw-Hill*, 2000.

[8] D. L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Trans. Communications 39*, 10, Oct. 1991.

[9] Neko v0.8 website.
http://lsrwww.epfl.ch/neko/

[10] NekoStat v0.8 website.
http://www.arcetri.astro.it/~falai/nekostat.html

[11] RFC 1305. Network Time Protocol (Version 3).
http://rfc.sunsite.dk/rfc/rfc1305.html

[12] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An operating system coordinated high performance communication library. In *Proc. High-Performance Computing and Networking (HPCN 97 Europe)*, 1997.

[13] P. Urban. Evaluating the performance of distributed agreement algorithms: tools, methodology and case studies. *Ph.D. Thesis, École Polytechnique Fédérale de Lausanne*, 2003.

[14] P. Urban, X. Defago, A. Schiper. Neko: a single environment to simulate and prototype distributed algorithms. In *Proceedings 15th International Conference on Information Networking (ICOIN-15)*, Feb. 2001.