Bridging the Gap between Simulation and Experimental Evaluation in Computer Networks

Marinho P. Barcellos,[*] Giovani Facchini, Hisham H. Muhammad
Guilherme B. Bedin, Paulo Luft
marinho@acm.org, {facchini, hisham.hm, gbedin,pauloluft}@gmail.com
PIPCA - Postgraduate Program in Applied Computing
UNISINOS - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950. São Leopoldo, RS. CEP 93022-000

Abstract

Simulation and experimentation are two complementary techniques for performance evaluation, each one of them having opposite characteristics and advantages. While one allows total control and abstraction in the experiment, the other provides greater detail and realism. Though ideally it would be desirable to perform both, it is hard to direct the required efforts to develop a performance evaluation twice, once over a simulator and then again on a real network. In this paper a new approach is explored through a tool called Simmcast Testbed, which allows one to execute, from a single codebase, experiments both in simulation and experimentation mode. A didactic example is discussed in detail, and the correlation of the simulated and experimental results is presented.

1. Introduction

There are three classic methods of evaluating the performance of computer networks and their protocols: *analytical evaluation*, *simulation*, and *experimentation* (also called *experimental evaluation*). These methods have been extensively explored in both academy and industry, and are well-understood by scientists and engineers. Analytical evaluation is very useful to determine general properties of a protocol, by measuring performance and cost under input parameter variation. However, it works well only for simple models, as they need to be expressed as a set of equations.

Simulation allows a variable degree of abstraction, from simple to detailed, depending on the model built and of the corresponding source code. As exemplified in our earlier work ([1]), simulation has the potential to allow one to gracefully increase its degree of detail, such that the simulation development process can result in protocol instances (e.g., sender, receiver, client, server, master, etc.) running on top of an "emulated network", ready to be moved to a "real network".

Experimentation consists of a series of test executions of a protocol over a physical network, and therefore can produce more realistic results. Nonetheless, because of the great influence of the topology, the operating systems and the specific settings adopted, results obtained are tied to a configuration and thus are hard to reproduce by other researchers. The logistics involved in realizing experiments, like obtaining accounts, gets in the way when experiments are of internet scale and involve multiple autonomous systems. Clearly, experimentation requires implementing the protocol, or at least a functional prototype. Each methodology has its own relative merits: none can be applied equally well to all situations.

Although discrete simulation is very common in the field of computer networks, the results produced, isolatedly, are not reliable enough. Limitations or common mistakes in using simulation, such as indicated in [4] and [3], may lead to distorted results. The same applies to analysis or experimentation. Particularly, in the latter, wrong values can be found due to unpredictable factors, background traffic and improper computer or network hardware configuration. Therefore, it would be advisable to employ all three methods complementarily, as part of a process to understand the workings of a protocol, and to obtain performance and cost measurements in regard to its main input parameters and desired network conditions. Pragmatically, it is likely that two methods combined will suffice; which two will depend mainly on the problem at hand and the available tools.

We would expect to find simulation accompanied by either analytical or experimentation. An attractive alternative for many cases would be to use simulation along with experimentation. To understand how frequently this occurred in computer network research, we examined over 600 papers from recent editions of highly selective scientific events on the field (namely, ACM SIGCOMM, IEEE INFOCOM, and IEEE ICNP). Results found confirm that simulation is very common, appearing in 67.4% of cases. The study also indicates that simulation is used by itself in 8.0% of cases and

31.1% along with experimentation.

We believe simulation does not appear more often with experimentation due to the implementation effort associated with each of these methods. Further, when they do appear in tandem, these methodologies are used separately, one after the other. This occurs because, among other reasons, the simulation code is different from the prototype code employed during experimentation.

It would be desirable that simulation and experimentation could be part of the same, integrated set, such that the researcher could seamlessly alternate between both methods. Clearly, not all types of work on protocols allow this situation; for example, a simulation support is required to provide different levels of abstraction, and to emulate underlying layers.

In this paper, we propose an extension for the Simmcast network simulator ([1, 7]) that promotes the use of simulation along with experimentation for application-level protocols. The rest of the paper is structured as follows. Section 2 presents Simmcast Testbed, including its interface and usage. A simulated reliable multicast protocol is described in Section 3. Although simple, it serves as a proof-of-concept. This protocol is employed in Section 4 to obtain simulation as well as experimental results. By measuring the correlation between simulated and actual results, we evaluate the power of our proposal. Section 5 closes the paper with final remarks and future work.

2. Simmcast Testbed

Simmcast Testbed is an Application Programming Interface (API) for the Java language, which isolates the programmer from both the peculiarities of the simulation environment and network library details. The API comprehends communication resources and a thread model, being composed of two implementations, according to the back-end employed: either the Simmcast framework or the actual Java APIs for network programming and threads.

The Testbed proposed is based on Simmcast, a simulation framework for protocol and distributed systems. Before presenting interface and implementations of the Simmcast Testbed API, we briefly review Simmcast, emphasizing its role as a simulation back-end.

2.1. Simmcast as a simulation back-end

Simmcast is an object-oriented framework for network protocol simulation ([1, 7]). Simmcast Testbed employs this simulation framework as one of the possible back-ends for its execution. The main properties of Simmcast are a modular architecture and a process-based simulation model, as explained below.

In the Simmcast architecture, classes correspond to building blocks that are dynamically connected to build the simulation environment. Hosts correspond to `HostNode` objects, and routers to `RouterNode` objects (both derived from `Node`, which represents a more abstract notion of node).

The process-based simulation model seems very adequate for the simulation of computing systems, since each process corresponds directly to an execution thread of the simulated program. In Simmcast, each process or thread is mapped as a `NodeThread` object. By allowing `Node` objects (and hence, `HostNode` and `RouterNode`) to be comprised of one or more `NodeThread` objects, multithreaded protocols may be designed in a natural manner.

These two characteristics make the Simmcast API quite realistic, similar to the actual communication and threading APIs in Java. However, at the same time, it allows more abstract simulations, through higher-level entities like `Node`. This intentional proximity to the APIs was a key factor in the development of Simmcast Testbed. It allowed the design of this new API to be restricted only by the properties of the real execution environment, free from simplifications imposed by the simulation environment.

2.2. Simmcast Testbed API

Simmcast Testbed is an API that has two implementations: `Testbed-Sim` and `Testbed-Exp`, which use as back-ends, respectively, the Simmcast framework and the Java thread and network APIs. Both are implemented through a Java package called `simmcast.testbed`.

The thread model of Simmcast Testbed follows the model adopted by Java, and is comprised of two classes: `Program` and `ProgramThread`. The class `ProgramThread` corresponds to a Java thread, and in fact, the set of methods offered is very similar. The class `Program`, on its turn, corresponds to an independent instance of a program, including the `main` method through which other threads are initiated. Consider, for example, a client/server application where the server has multiple threads for the concurrent handling of requests. In this case, client and server correspond to two independent instances of `Program`, whereas the server threads would be multiple instances of `ProgramThread`. Another example is the structuring of a complex protocol in a set of threads with synchronous behavior, separating in a node the roles of transmission and reception of packets; in this case there would be one instance of `Program` and two of `ProgramThread`.

The primitives of Simmcast Testbed for sending and receiving data are in an intermediary level of abstraction between the Simmcast and Java models. In Simmcast Testbed, class `ProgramThread` directly provides three primitives for sending and receiving data: `send()`, `receive()` and `receiveMulticast()`. The `send()` method receives as an argument a byte vector and the methods `receive()` and `receiveMulticast()` return an object `DataPacket` containing the byte vector received.

Management of sockets is done implicitly in `Testbed-Exp`: when a sending or receiving primitive is called with an address which was not specified previously, a new socket is opened internally. Another important abstraction is the management of multicast groups: in the `Testbed-Exp` implementation, when sending to a new group, the request `joinGroup()` is performed transparently; in the `Testbed-Sim` implementation, the building block `Group` of Simmcast is employed. In the `Testbed-Exp` implementation, data is transmitted through ordinary datagrams.

While IP addresses in Java are codified through `InetAddress` objects, in Simmcast nodes and multicast groups are identified more simply through numeric identifiers. In Simmcast Testbed, a class `NetworkAddress` abstracts network addressing. Methods like `getLocalHost()` are made available such that the addresses are always specified in the form of `NetworkAddress` objects.

Asynchronous timers are a resource commonly used in the description of protocols. Simmcast possesses a callback resource through the `onTimer()` method, which simplifies the specification of timers. This resource was reproduced in Simmcast Testbed.

Because of these properties, the API of Simmcast Testbed keeps a strong correspondency with the network and threading programming model provided by Java, as well as with the programming model employed with Simmcast. In fact, this similarity can be observed in Table 1, which summarizes and compares the methods.

2.3. Simmcast Testbed Usage

The choice of the execution model (simulated or real) happens at execution time, when specified to the *Java Virtual Machine* (JVM) which copy of the package `simmcast.testbed` should be imported. Thus, it is possible to alternate between the `Testbed-Sim` and `Testbed-Exp` modes without recompilation.

In the "real" execution, each instance of `Program` corresponds to an independent JVM, possibly executing in distinct computers. In the simulated run, a single instance of the simulator will create all instances of `Program` and execute them in a single JVM. Consequently, the way experiments are started differs depending on the execution mode, and must be prepared separatedly. For the initialization of the `Testbed-Exp` mode, Simmcast Testbed offers a `Main` class, which has a static method `main` and receives a `Program` class as a parameter. In the `Testbed-Sim` mode, the initialization of `Program` objects is done through the plain text file that describes the simulation, constructing the simulation scenario using dynamic class loading. Apart from the initualization, all protocol code used in modes `Testbed-Exp` and `Testbed-Sim` is identical.

When using Simmcast Testbed, some care is necessary to ensure portability between the two modes of execution. The main precaution is to avoid accessing directly the underlying APIs (either the network and threading APIs of Java, or the Simmcast APIs), or else it is likely that the resulting code will work solely in one of the two modes. Other important measures refer to the peculiarities of the real and simulated environments. In the real environment, static variables can be used (e.g., to allow sharing among threads), since each station executes its program in a separate JVM. In contrast, in the simulated mode all `Program` objects would erroneously share the same variable.

The intentional sharing of variables, although a common technique in simulation, causes a portability problem for programs developed initially for `Testbed-Sim` mode and later executed in `Testbed-Exp` mode. Global variables can be a useful resource in simulations, because they can be used for the declaration of global input simulation parameters as well as collection of global statistics. When using Simmcast Testbed, per-node global data can be stored as instance variables of `Program` objects.

It is important to note that a big difference between the Testbed-Sim and Testbed-Exp environments in the presence of a global synchronized clock in the former. If the code assumes the synchronicity of the clocks of the various nodes into account, the experiment is likely to fail when moved from Testbed-Sim to Testbed-Exp. Therefore, it is advisable to assume an asynchronous network even when working in simulation mode.

3. Case Study

In this section, we present a case study that demonstrates the use of Simmcast Testbed. Since the purpose is to present a development methodology, we employ a simple example, so that it can be presented in its entirety. The example employed is a reliable multicast that follows the "Stop-and-Wait" approach. An implementation of this protocol is provided as one of the examples of Simmcast[1]. Through Simmcast Testbed, the example is adapted to be executed both in the simulator and in a real network.

[1]available at `http://www.unisinos.br/~simmcast`.

Simmcast Testbed	Simmcast	Java
ProgramThread	NodeThread	Thread
Program	HostNode	*main program*
DataPacket	TransportPacket	DatagramPacket
NetworkAddress	Node.networkId (int)	InetAddress
Clock.getDate()	Network.simulationTime()	System.currentTimeMillis()
ProgramThread.send()	Node.send()	DatagramSocket.send()
ProgramThread.receive()	Node.receive()	DatagramSocket.receive()
ProgramThread.receiveMulticast()	Node.receive()	MulticastSocket.receive()

Table 1. Correspondency between methods and classes of Simmcast Testbed and the equivalent ones in Simmcast and in the Java Language.

3.1. Transmission protocol

Experiments with the Stop-and-Wait reliable muticast protocol are easy to reproduce. This (1-*N*) model features a sender and *N* receivers in a destination group, whose identity is known. Receivers receive packets transmitted by the sender and reply to each received packet with an ACK back to the sender. The role of the sender is to transmit a packet and wait for one or more ACKs from each receiver in the destination group. Losses are detected at the sender by means of a timeout: the sender transmits a data packet and starts a regressive timer, which can be either cancelled if all ACKs exepected are received in time, or it can expire, in which case the sender assumes there has been a loss of data or ACKs and retransmits the packet. All packets have an increasing sequence number, and the sender can only transmit packet $seq + 1$ after all ACKs for seq have been received.

Each receiver node is implemented as an object of the `SinkNode` class. The latter is a subclass of class `Node` of Simmcast. It contains a single execution thread, `SinkThread`, through which the protocol logic is described. Analogously, the sender is implemented as an object of the `SourceNode` class which contains the execution thread `SourceThread`. In the configuration file, a network with a star topology is specified and indicated to sender the identity of the receivers members of the group.

3.2. Adaptation to Simmcast Testbed

The preparation for the execution of the protocol in Simmcast Testbed is comprised of two steps: first, it is necessary to adapt the program for the Simmcast Testbed API. Second, it is necessary to adapt the code to remove functionality based on assumptions that are valid only for simulation, such as shared variables. The first step is straightforward. As presented in Section 2.2, the API is intentionally similar to both the simulator API as well as the Java API, to ease conversions in both ways. During this step, the program may be tested in the simulator employing the `Testbed-Sim` mode of execution.

The second step consists in removing from the experiment simplifications adopted during simulation model development which would prevent the operation in a real network, including potential measurements in interval times based on the assumption of a global simulation clock. This step requires an analysis of the characteristics of the experiment in question. Considering the receiver presented in this example, we only need to modify the access to the API an employ a serializer to send data, because in the new API the primitive takes an array of bytes instead of an object. Thereafter, the receiver can be executed in both `Testbed-Sim` and `Testbed-Exp` modes. The changes required to the sender for this model are analogous to the receiver, since it does not make strong simulation-based assumptions such as shared clocks. Another small change required is the passing of parameters for initialization, since in the simulator this is done through a Simmcast script and in the real network through the command line.

Figure 1 compares the main execution loop of the `SourceThread` class in the original version and in the adapted one. Note that the changes presented here are restricted to API syntax and the usage of the `Serializer` class to transform objects into byte arrays, because in the simulator packets carry arbitrary Java objects, with size as an abstract property, and in the Testbed it is necessary to specify the actual sequence of bytes that will flow through the network. In the simulator, packets are identified by an attribute called `type`. To provide similar information when running over the network, a `DataObject` class is used, which contains the sequence and type information.

In order to build a simulation model that is close to the real implementation and achieve results in simulation that resemble the ones in the real network, we first monitored the network and determined its main properties. In particular, we assessed the mean and standard deviation for end-to-end latency and average loss rate. Along with bandwidth

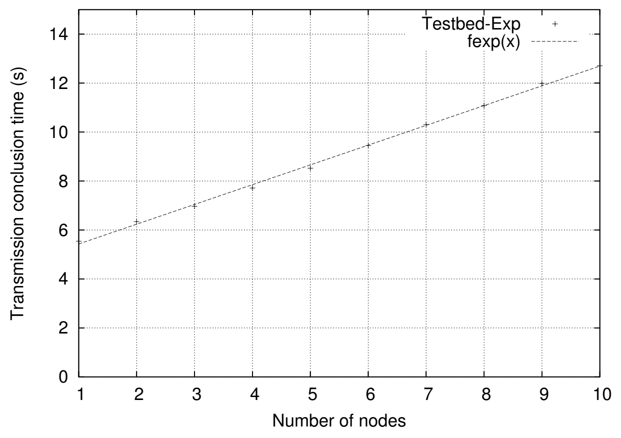
```
int source = sourceNode.getNetworkId();
int destination = sourceNode.gid;
PacketType packetType = new PacketType("DATA_PACKET");
int size = 1;
for (int i = 1; i <= sourceNode.numPktsToSend; i++) {
   Integer msg = new Integer(i);
   boolean isAckSetComplete = false;
   TreeSet ackList = new TreeSet();
   while (!isAckSetComplete) {
      send(new Packet(source, destination, packetType, size, msg));
      while (!isAckSetComplete &&
         (numTimeouts + ackList.size()) < sourceNode.gids.length) {
         Packet reply = (Packet) receive(sourceNode.timerLength);
         if (reply != null){
            Integer seq_r = (Integer) (reply.getData());
            if (seq_r.equals(msg)) {
               int receiver = reply.getSource();
               ackList.add(new Integer(receiver));
               if (ackList.size() == sourceNode.gids.length) {
                  isAckSetComplete = true;
               }}}}}
}
```

(a) Implementation of `SouceThread`
for the Simmcast simulator.

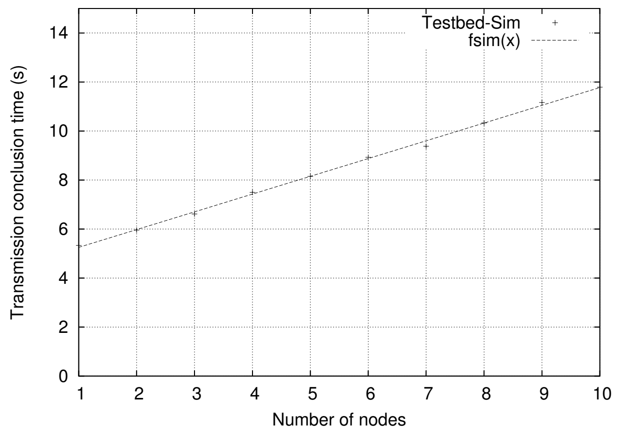
```
Clock clock = new Clock(sourceNode);
NetworkAddress destination = sourceNode.gid;
int size = 1;
for (int i = 1; i <= sourceNode.numPktsToSend; i++) {
   Integer msg = new Integer(i);
   boolean isAckSetComplete = false;
   TreeSet ackList = new TreeSet();
   while (!isAckSetComplete) {
      byte[] dataToBeSent = null;
      DataObject dataObject = new DataObject();
      dataObject.seq = msg.intValue();
      dataObject.type = "DATA_PACKET";
      dataToBeSent = Serializer.serialize(dataObject);
      send(dataToBeSent,dataToBeSent.length, destination, sourceNode.port);
      while (!isAckSetComplete &&
         (numTimeouts+ackList.size()) < sourceNode.receivers.size()) {
         DataPacket reply = receive(port,timeOut,dataLength);
         if (reply != null) {
            DataObject dataInPacket = null;
            dataInPacket = (DataObject) Serializer.rebuild(reply.getData());
            int seq_r = dataInPacket.seq;
            if (seq_r == msg.intValue()) {
               ackList.add(reply.getSource().toString());
               if (ackList.size() == sourceNode.receivers.size()) {
                  isAckSetComplete = true;
               }}}}
   }
```

(b) Adaptation of `SourceThread`
to Simmcast Testbed

Figure 1. Comparison between original simulator code and the code adapted to the execution in both simulator and real network.



(a) Experimental (real) execution



(b) Simulated execution

Figure 2. Results of protocol execution in terms of transfer time.

and physical topology, these parameters describe network in terms of its connectivity and link properties. Using the protocol implementation, we measured the total time required to complete execution, delays associated with scheduling and processing overheads from incoming and outgoing packets. To model the delays associated with the processing of packets in the operating system network stack, we employed two simulator primitives: `setReceiveTime` and `setSendTime`. The link delay parameters were modeled through the average latency observed in experiments with the real network.

4. Results

Each single run consisted of a simple session in which the sender reliably transmited 10,000 packets to the group

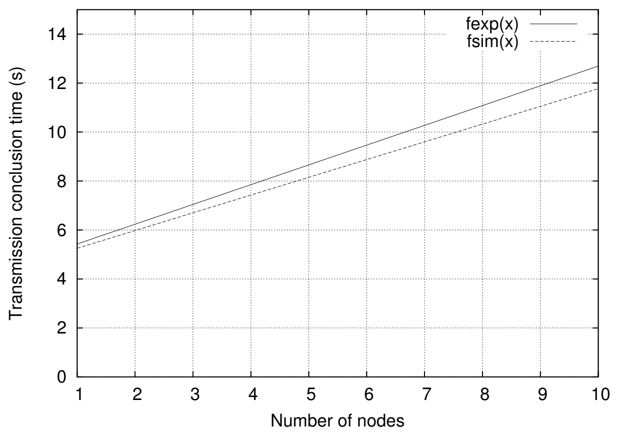


Figure 3. Comparison between the functions of linear adjustment for experimental and simulated executions.

of receivers. The size of the destination group varied from 1 to 10 receivers. For each combination of group size and execution mode, 10 iterations were run, totalling 200 experiments.

The experiment with `Testbed-Exp` was performed in a controlled environment. The tests were run on a cluster of 11 Linux machines connected through a Gigabit network (all machines connected to a Gigabit Ethernet switch). The machines were Dual Xeon 2.4GHz with 1GByte of RAM running Gentoo Linux kernel 2.6.10. One machine ran the sender, whereas the rest ran the receivers, one per machine.

To reproduce this scenario, the configuration file that describes a simulation scenario was set to represent a star topology. All nodes of type `SinkNode` (receivers) and `SourceNode` (sender) were connected to a node of type `DefaultRouterNode` (switch). The properties of links connecting nodes mirrored the real scenario: 1Gbps bandwidth, negligible loss rate (we observed less than 0.001%) and latency described by a Normal distribution with average 0.095ms and standard deviation 0.02.

The results of executions in `Testbed-Exp` and `Testbed-Sim` modes are presented in Figures 2(a) and 2(b), respectively. In both cases, they show the points from all iterations and the linear adjustment for each execution mode. The linear adjustment function of execution in `Testbed-Exp` is $fexp(x) = 4.62199 + 0.807406x$, and the adjustment of execution in `Testbed-Sim` is given by $fsim(x) = 4.52811 + 0.724613x$ (all values are given in seconds).

The adjustment of point in a function aims to represent the behavior of the protocol being modeled. When comparing the adjustments obtained between executions in simulated and experimental modes, it becomes possible to verify if the behavior shown by the execution of a program in both modes are equivalent.

The two adjustment functions are presented together in Figure 3. We can observe clearly that it is possible to obtain values that are very close in simulated and real executions and, mainly, that the behavior of the protocol in both modes are consistent between each other. The small distance between the lines was expected, since modeling the behavior and overhead induced by scheduling is a challenging task and depends heavily on the operating system installation and hardware architecture used.

5. Concluding Remarks

Simmcast Testbed is being actively used in the investigation of high-performance transmission protocols, mechanisms for large-scale online games, and secure computing in P2P networks. Testbed acts like a link between two of the main techniques employed in performance evaluation in the fields of computer networks and distributed systems: simulation and experimentation. We presented in this paper not only a tool, but a methodology for the development of research in this area, in which the controlled environment of a simulator can be used during implementation and debugging, and the richness of detail of a live network environment can be used to obtain precise results. The main advantage of this methodology is the possibility of taking turns between the two environments on an as-needed basis, without having to port the code again. This is only possible due to the fact that Simmcast Testbed is a "transitional" API, sitting between the levels of abstraction of a simulator and a network programming library.

The approach proposed in this paper is novel. It cannot be confused with network emulation facilities, such as the ones offered by certain simulators. In VINT ns-2, emulation ([10]) means to allow a simulation to be executed in a machine that will generate, as events in its simulation, the sending of packets in the real network; conversely, a simulation in another machine could receive packets and interpret the reception of such packet as an entry (event) in the simulation. Note that the simulation clock differs from the physical clock time, since there are two distinct simulations involved, with independent clocks and discrete events. For NIST ([9]) and Delayline ([8]), emulatiton means to execute applications in a *testbed* comprised of a set of workstations modified to add delays, packet losses and other desired conditions in a network during an experiment.

One similar initiative is Netbed ([14]). It is an environment that offers a common set of abstractions for different types of links and nodes to make an environment that combines simulation, emulation and experimentation in a real network. Its design generalizes resources and mechanisms in common abstractions applicable to a variety of realizations of emulation, simulation and experimentation. Netbed is based on NS-2 and offers to the users the access to a physical testbed and nodes distributed through a Web interface. Netbed, unlike Simmcast Testbed, does not allow the same protocol code or application be transparently migrated.

The approach proposed here can be compared in regard to the methods of simulation and experimentation, when employed isolatedly. Individually, it is expected that each one requires less effort than when developing through Simmcast Testbed. Without Testbed, the amount of work required to "convert" a simulation into a working prototype will vary from case to case, depending on the target, the language employed, and the way the protocol was modeled in the simulation. The higher the level of detail included in the simulation, and allowed by the simulator, the smaller should be the effort.

In terms of functionality, the Testbed API imposes certain restrictions in regards to what can be done by the protocol code. Noticeably, the only existing support currently is the transmission of messages by means of UDP datagrams. That is, there is no support for TCP streams not more sophisticated communication functionality, such as Remote Method Invocation (RMI). Some of these restrictions are required for the implementation to run properly in both environments, simulated and real.

In terms of performance, the `Exp` version of the protocol will present results that are inferior to the ones in a native Java implementation, due to the overhead induced by the thin layer `Testbed-Exp` that implements the API. There is no data at this point to quantify such overhead, but since it consists of a constant increase in processing time, the only adverse effect should be a constant offset in the observed results, without affecting their overall trends. Also, since the Testbed API is modeled to resemble the Java API, once the simulation and experimentation stages are completed, the codebase can be used as a foundation for a native implementation.

References

[1] M. P. BARCELLOS, H. MUHAMMAD, and A. DETSCH, "Simmcast: a Simulation Tool for Multicast Protocol Evaluation", XIX Simpósio Brasileiro de Redes de Computadores (SBRC 2001), Anais, SBC, Flps., 21-25 Maio 2001.

[2] L. BRESLAU et alli. "Advances in Network Simulation". In IEEE Computer, volume 33, n. 5, pp. 59-67, May 2000.

[3] J. BYERS, G. HORN, M. HANDLEY, M. LUBY, W. SHAVER, and L. VICISANO. "More Thoughts on Reference Simulations for Reliable Multicast Congestion Control Shemes". Notes from a meeting at Digital Fontain, August 8, 2000.

[4] R. JAIN. “The Art of Computer Systems Performance Analysis”. John Wiley & Sons, 1991.

[5] I. KEIDAR, R. KHAZAN, N. LYNCH & A. SHVARTSMAN, “An Inheritance-Based Techique for Building Simulation Proofs Incrementally”, ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 1, January 2002.

[6] N. LYNCH, “Distributed Algorithms”, Morgan Kaufmann, San Francisco, 1996.

[7] H. H. MUHAMMAD, M. P. BARCELLOS, “Simulating Group Communication Protocols Through an Object - Oriented Framework”, 35th Annual Simulation Symposium (SS2002), Proceedings, IEEE (New York), San Diego, 14-18 April 2002.

[8] D. B. INGHAM, G. D. PARRINGTON, “Delayline: A Wide-Area Network Emulation Tool”, Computing Systems, v.7, n.3, 1994.

[9] NISTNET, Project site, `http://dns.antd.nist.gov/itg/nistnet/`.

[10] NS-2, Project site with online documentation about NS-2, `http://www.isi.edu/nsnam/ns/ns-emulation.html`.

[11] NS-2, Project site, `http://www.isi.edu/nsnam/ns/`.

[12] SIMMCAST, Project site, `http://www.inf.unisinos.br/~simmcast`.

[13] B. WHITE, J. LEPREAU, S. GURUPRASAD, “Lowering the Barrier to Wireless and Mobile Experimentation”, Proceedings of the First Workshop on Hot Topics in Networks (HotNets-I), October 2002.

[14] B. WHITE et alli. “An Integrated Experimental Environment for Distributed Systems and Networks”, Proceedings of the 5th Symposium on Operating Systems Design & Implementation, pp. 255-270, December 2002