Simmcast: a Simulation Tool for Multicast Protocol Evaluation

Marinho Barcellos Hisham Muhammad
André Detsch

PIPCA- Programa de Pós-Graduação em Computação Aplicada
C6 - Centro de Ciências Exatas e Tecnológicas
UNISINOS - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950 - São Leopoldo, RS - CEP 93.022-000 - BRAZIL
{marinho,hisham,detsch}@exatas.unisinos.br

Abstract. This paper describes Simmcast, a process-based discrete-event *simulation framework* that helps the design and evaluation of multicast protocols. Simmcast, based on Java, allows simulated protocols to be specified using a multi-threaded, object-oriented framework; building blocks are combined in order to create new protocol simulation experiments. To help validating Simmcast, two reliable multicast protocols used in a well-known analytical study were simulated and the results achieved compared the ones produced by Simmcast. In Simmcast, users are first presented an easy-to-understand, abstract multicast protocol simulation model, and then can gradually increase the level of detail in their simulation. Unlike existing tools, Simmcast provides a flexible simulation model which is dedicated to the design and evaluation of *multicast protocols*.

Resumo. Este artigo descreve Simmcast, um *framework* para simulação discreta de eventos baseada em processos que contribui para o projeto e avaliação de protocolos multicast. Simmcast, baseado em Java, permite que protocolos simulados sejam especificados usando um framework *multi-threaded* orientado a objetos; "blocos de construção" são combinados para criar novos experimentos de simulação de protocolos. Com o objetivo de validar o Simmcast, dois protocolos multicast confiáveis usados em um consagrado estudo analítico foram simulados, e seus resultados comparados com os resultados obtidos com Simmcast. Simmcast oferece um modelo abstrato para simulação de protocolos multicast que é fácil de entender e manipular. Usuários podem então gradualmente aumentar o nível de detalhe nas suas simulações. Diferentemente de ferramentas existentes, Simmcast oferece um modelo de simulação flexível que é dedicado ao projeto e avaliação de *protocolos multicast*.

Keywords: performance evaluation, multicast protocols, simulation, middleware.

1 Introduction

Simulation has been a powerful tool to help designing and evaluating communication protocols. Other ways of gaining knowledge about a protocol are analytical evaluation and experiments (e.g., [20] and [15], respectively). These three ways are complementary, and may represent different parts of the same design process. Analytical evaluation is very useful as it can be used to demonstrate the general impact of protocol input arguments (such as window size), number of participants, or changes in specific network conditions (such as loss rate) on protocol performance. However, it works well only for simplistic models, since results are obtained by changing the input arguments in formulae (the complexity lies in the number of variables considered). Analytical evaluation is useful to either determine overall trends in abstract protocol behavior, or to prove formally a protocol's correctness.

In contrast to the simplified view of analysis, practical experiments are based on a set of test runs of a protocol communicating over an actual network, and thus produce the most accurate results. Nonetheless, because of the great influence of topology, systems and configurations involved, the results collected are bound to a specific setup, and are difficult to reproduce elsewhere. This is particularly true when scalable multicast protocols are to be evaluated, for it is impractical to perform a large-scale experiment in a wide-area network. A considerable drawback of this approach is that it requires the protocol to be implemented before its general workings can be tested.

Simulation lies at intermediate level, between analytical evaluation and practical experiments. It allows the protocol designer to adjust the level of detail, retaining only desired features. Further, simulation has the potential to allow the gradual increment of detail, so that the process results in a protocol running over an emulated network, ready to be moved to an actual, live network.

This paper describes Simmcast, or Simulation of Multicast[1]. Simmcast is a simulation framework that allows protocols to be easily defined by a combination of basic building blocks (the idea of using building blocks to design multicast protocols is being pursued elsewhere; for example, see [21]). It is object-oriented and supports multi-threaded multicast protocols. The main difference between unicast and multicast protocols is the concept of *group*. In reliable multicast protocols, agents need the ability to join and leave groups, and to send packets to one or more of unicast and multicast addresses. Support for multicast communication are builtin in the simulator, affecting its input arguments, its queuing model, and its output metrics and traces.

Simmcast is based on Java and JavaSim ([13]), a simulation toolkit whose aim is to emulate the Simula language ([3]) facilities in Java. JavaSim, and thus Simmcast, follow the same process-based, object-oriented, discrete event model introduced by Simula. Simmcast indirectly inherits

[1] Simmcast also recursively stands for "Simmcast is much more concise and structured than..."

all the advantages of Simula: its simulation facilities, classes and libraries have a considerably experienced user community which have found them to be successful for a wide variety of simulations, including those of communication protocols.

The paper is organized as follows: Section 2 provides an overview of Simmcast, while Section 3 illustrates the use of Simmcast and validates its results. Section 4 discusses related work, and Section 5 concludes the paper.

2 Simmcast Overview

Simmcast is tailored to all kinds of multicast protocols: routing multicast protocols, transport-level reliable/semi-reliable multicast protocols, and distributed applications whereby agents communicate over multiple, potentially overlapping multicast groups. The next subsections provide an overview of Simmcast, beginning with its Application Program Interface.

2.1 Application Program Interface (API)

As already mentioned, Simmcast is a discrete simulation *framework* based on the Java programming language. The use of frameworks increase software reuseability, which can lead to advantages like reduced development effort and more robust code through multiple reuse and refinement of the framework. Simulation seems particularly appropriate for frameworks, since substantial part of the code can be reused between simulation experiments. The concept of frameworks has been applied to network simulation (e.g., [6], [14]), to the development of network protocols (e.g., [8], [19]), and also to build a multicast service (e.g., [18]). The Java Language, on its turn, has been recently a popular choice for simulation software (e.g., [9]) and communication protocols (e.g., [11], [1]).

The main idea behind the proposed multicast simulation framework is to provide an API with the typical communication and timer operations, as well as suitable (concurrent) software architecture for designing and evaluating multicast protocols. To get a simulation running, the user needs to add or extend classes or interfaces of the framework according to the specific protocol and configuration being evaluated, but still uses most of the framework's functionality without having to reinvent that functionality (e.g., sending of unicast and multicast messages).

Simplicity of use is one of the main goals of Simmcast. The use of simpler, synchronous thread modules in protocol building is encouraged. There are only ten primitive operations provided to the user:

- `send`: send a packet to a given destination address, either unicast or multicast;

- `receive`: request the receipt of a packet, blocking until a packet is available;
- `tryReceive`: try receiving a packet, returning "null" if no packet is available;
- `join`: receiver joins a given multicast group;
- `leave`: receiver leaves a given multicast group;
- `setTimer`: configure timer to expire in a given time in the future;
- `cancelTimer`: cancel an existing timer;
- `onTimer`: method invoked when a timer expires;
- `sleep`: puts the current thread to sleep for a given time;
- `wakeUp`: wakes up another thread that may be sleeping.

2.2 Building Blocks

In Simmcast a simulation is described by combining a set of building blocks, providing additional code when required. Building blocks themselves are made up mixing two basic components: *processes* and *queues*. Processes are active objects that correspond to one thread of execution. Processes add and remove objects to and from queues (in fact, ordered lists). Queues are used mostly to model packets in transit. Below, the building blocks are presented; for each building block, there is a corresponding class.

Node. Nodes are the fundamental interacting entities, and uniquely identified by an integer. Depending on the desired level of abstraction, nodes can represent a protocol agent in a host, a router, or one of many interacting entities in a host/router. A node will contain one or more threads of execution. Suppose x and y are nodes. Every x has one send queue for each outgoing connection (queue denoted as $SQ_{x,y}$, when sending from x to y), and a single receive queue (denoted as RQ_x), to whom all packets arriving (at x) are added to. Both SQs and RQ have finite length; SQ is served according to link bandwidth, while RQ is served according to the node thread that receives packets. Send and receive times (T_{send} and T_{recv}, respectively) are the times taken to send and receive packets, and can be configured by the user. Each x has also a timer queue, TQ_x, to hold asynchronous events.

Link. Nodes are connected by links. Link properties are bandwidth, packet loss probability (denoted as ε), and propagation delay (T_{prop}). T_{prop} can be fixed or drawn from a user-chosen random distribution stream. Each link has an associated link queue ($LQ_{x,y}$) to hold packets being propagated from x to y. Links in Simmcast can represent a physical link,

a channel which combines network properties of several physical links (like in [2]), or a local message queue.

Group. The concept of group is paramount to multicast protocols. The membership of a group can be defined by the user via the simulation description file that describes the experiment. Alternatively, nodes, through the protocol software associated, are able to dynamically join or leave groups, according to the protocol logic.

Network. A network is an arbitrary combination of nodes and links. Network connectivity is described through a square matrix of size $N \times N$, denoted as L (of links). An element $L_{x,y}$ corresponds to the unidirectional link from node x to y. No specific routing scheme is enforced by the simulator, so that different kinds of routing schemes can be added by creating/using routing nodes (by specializing the `Node` class with some routing logic). Consequently, when a node x sends to another node y, x must be directly connected to y ($\exists L_{x,y}$). Likewise, when a node sends to a group g, x must be directly connected to all elements of g (for every $y \in g$, $\exists L_{x,y}$).

Packet. Packets are the unit of communication between any two or more nodes. The packet class contains the minimal attributes required by a packet in Simmcast, and for new protocols, inheritance is used to define new packet types.

Figure 1 depicts the internal structure of a node and the packet flow through its queues; there are three nodes, x, y and z, but only y is fully represented. When y sends a packet to z ($y \neq z$), the packet is enqueued in $SQ_{y,z}$, space allowing (otherwise the packet is discarded). The thread of y that invoked `send` remains blocked for T_{send} time, before control returns. The packet moves from $SQ_{y,z}$ to $LQ_{y,z}$ according to packet size and bandwidth. Note that by default $LQ_{y,z}$ has infinite size. The packet is then kept in $LQ_{y,z}$ for T_{prop} time, and is subject to loss according to the loss probability of the link. When a packet arrives at z, it is moved from $LQ_{y,z}$ to RQ_z, again space allowing. The packet remains in RQ_z for an arbitrary time, and leaves RQ_z according to `receive` and `tryReceive` operations issued by protocol logic of node z.

To complete the node specification, there are two building blocks that usually correspond to operating system resources, *threads* and *timers*.

Thread. Threads simplify a protocol because they allow the developer to model the architecture as a set of simpler interacting synchronous entities. More complex protocols or agents (like servers) can be simulated with threads that are dynamically spawn on demand to handle incoming messages (one thread per message). In Simmcast each process maps into a thread.

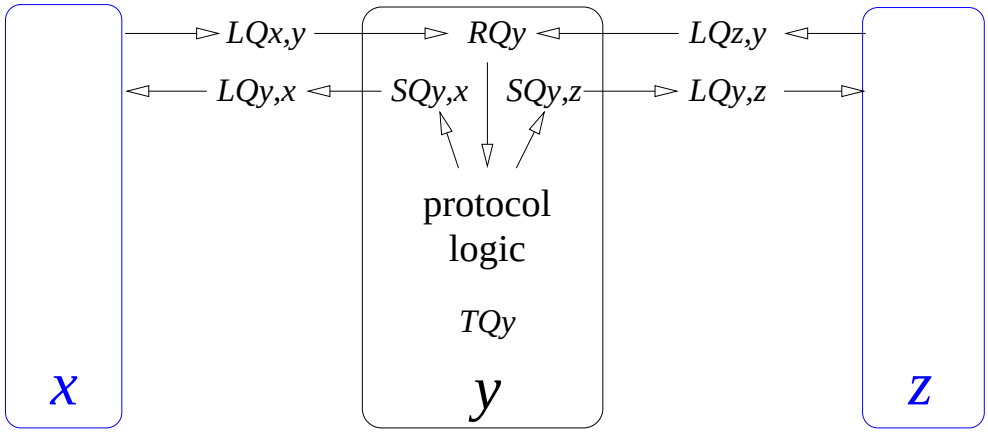

Figure 1: Illustration of node structure; *x*, *y*, and *z* are nodes, but only *y* is fully represented.

Timer. Asynchronous events can be implemented using timers. There are many cases of asynchronous events in protocol software, being timeouts the most common ones. Timeouts are used, for example, to detect packet losses. Also, permanent, consecutive timeouts allows a node to suspect that a network partition has occurred. Timers can also be used to implement periodic behavior, like TCP[2]. Timed events can be controlled by simply overriding the `onTimer` method of the `Node` class.

The combination of the previous elements is used to build the configuration for an experiment. Finally, there are other building blocks that are meant to help the design of protocols; the two most important ones are `Queue` and `SlidingWindow`. Queues are an important structure in protocol software, for protocol layers or cooperating modules can be expressed in terms of producer/consumer models that communicate over a common queue. Sliding windows are a fundamental mechanism for reliable protocols, being used to control packet transmission and confirmation of receipt; for example, TCP error, flow and congestion controls are all built around the concept of a sliding window. Sender-initiated reliable multicast protocols can benefit from a sliding window that can be extended to work with multicast groups.

2.3 Output (metrics)

Simulation allows protocols to be evaluated according to some given metrics. The most popular examples are throughput and network cost. Throughput is usually defined as the amount of user data transmitted over time required to (reliably) transmit it[3].

Network cost can be simply defined as the amount of bandwidth required in order to (reliably) complete the transmission of all the data. In unicast communication, such definition can be

[2] TCP implementations deal with events periodically at every 200 and 500ms intervals.
[3] in case of reliable transmission, also called *goodput*.

straightforwardly transformed into a formula. However, it is slightly more complex to evaluate network cost with multicast communication, due to the replication of packets that occurs as these traverse a multicast distribution tree. Note that a packet might be transmitted, in general, either via one or more unicast transmissions, or via a single multicast transmission, depending whether the packet is intended to the entire group or to a subset (e.g., in case of a retransmission a reliable multicast protocol must perform). The former approach sends the same packet several times, but only to those interested in receiving a copy of that particular packet, while the latter approach sends a single copy but to all receivers, regardless of interest in the packet. Irrespective of the unit chosen (packets, bytes or bits), there are at least three different ways of assessing network cost.

The first way to measure cost is to count the amount of packets *transmitted* by senders (data and control) and receivers (control). In this model, a unicast transmission costs as much as a 1-to-N multicast one, which is clearly untrue for any $N > 1$, considering the extra bandwidth (in $N-1$ links) and processing times required (at $N-1$ receivers). The second way distinguishes the cost between unicast and multicast by counting the number of packets *received*. In this model, a packet that is sent via multicast will cost N (receptions), while every unicast packet will cost only 1 (reception). Therefore, unnecessary use of multicast is penalized, as desired. However, this model does not consider the processing cost required to send the multiple copies of the same packet (when sending to a subset of the group): sending N copies of a packet represents the same cost (N receptions) of sending a single copy by multicast (also N receptions). This is clearly undesirable.

The above approaches are limited because they overlook network topology. The third way to assess cost is to sum all packets, bits or bytes transported through all communication links (data and control) that make up the topology. A protocol that sends all packets via multicast will be accounted for that, as well as a protocol that only sends via multiple unicasts. Simmcast provides ways to the user to collect such data and consolidate it.

Finally, there are other metrics that may be of interest, but it depends on the particular protocol or application considered. Examples of metrics are the number of congestion losses experimented in a bottleneck router when a congestion control mechanism has been active, or the mean time until a receiver can recover from a packet loss while receiving real-time traffic (e.g., audio or video).

Besides output metrics, trace files are an essential feature of protocol simulators. They allow a protocol behavior to be investigated similarly to what would occur with a sniffer in a live network. As Simmcast is based on discrete-event simulation, it allows events to be recorded on trace files. In Simmcast, all events are considered to be inclusion and/or removal of an element to/from its queues (SQ, LQ, RQ, TQ). For example, when a packet is sent, the following events will take place: adding a packet to SQ, moving the packet from SQ to LQ, moving the

packet from *LQ* to *RQ*, and removing the packet from *RQ*. Timer events are also represented by enqueing/dequeuing objects to/from *TQ*. Besides its native trace format, Simmcast can generate trace files in the VINT nam ([7]) format, so that multicast protocols simulations can be later visualized using this network animator.

2.4 Running a Simulation

To run a simulation, the user must specify three things:

1. the simulated protocol: a new protocol can be built by combining existing building blocks and specializing `Node` and related classes. There may be one kind of agent (e.g., peer group application), two kinds (e.g., sender, receiver, or client and server, or master and slave), three kinds (e.g., sender, proxy, receiver), and so on.
2. the network topology: instantiate nodes and define their connectivity. Different levels of network representation are possible. The default for Simmcast is the more abstract level: the network is a set of direct point-to-point connections between nodes. The topology is a single-level multicast tree. A higher level of detail is obtained by making part of the nodes act like routers, as previously mentioned.
3. the protocol topology: how to specify (input) the protocol topology (agent allocation) to network previously defined.

The latter two steps are done by preparing a simulation description file. The input arguments laid out in Subsection 2.2 can be configured through such file. For example, users can set the times of T_{send} and T_{recv} of each node to be a given value or draw from a given random distribution, the sizes of *SQ* and *RQ* (they *can* be limited to a number of packets), and the properties bandwidth, delay and error rate of each link in matrix *L*.

3 Example Protocol Models

3.1 Modeling of a Simple Multicast Protocol

This section illustrates a simple, yet nearly functional, reliable multicast protocol modelled using Simmcast. Its purpose is to illustrate the use of threads and object-orientation when describing a multicast protocol[4]. The chosen protocol is FF - Full Feedback, described in [2]. Figure 2 presents a diagram with its main classes.

[4]Due to space restrictions the source code cannot be included in the paper; the interested reader should refer to `http://www.inf.unisinos.br/~marinho/Research/simmcast.htm`.

The *source* node is an instance of the `FFSource` class, which in turn extends `Node`. It contains two threads, `FFSourceSender` and `FFSourceReceiver`. For each transmission or retransmission, `FFSourceSender` sends a packet to the multicast group address and sets a timeout using the `setTimer` facility. Retransmissions are scheduled in the overridden `onTimer` method, called whenever a previously set timeout expires. Timeouts may be canceled when `FFSourceReceiver` receives at least one ACK from every *sink* node. On the other hand, the sink nodes (instances of the `FFSink` class) are single-threaded. They block waiting for packets using the `receive` method, and, as each packet arrives, a corresponding ACK is transmitted.

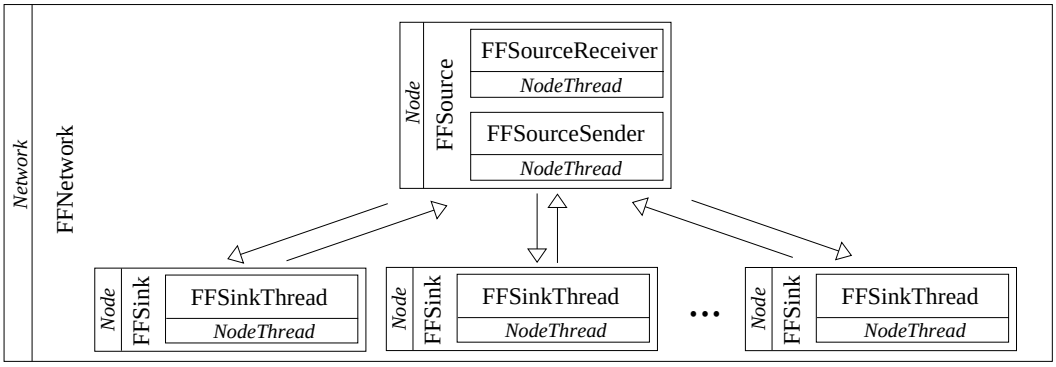

Figure 2: Diagram with modules for the FF protocol.

3.2 Modeling of two Abstract Reliable Multicast Protocols

This section compares the results of an experiment using Simmcast with results taken from a well-known analytical study of the scalability of reliable multicast protocols ([20]). The aim of *that* study was to determine the maximum packet processing rates attainable at sender and receivers for *sender-initiated* and *receiver-initiated* models. So, the study employed only abstract network and protocol models, with many implications: the sender always re-multicast to the entire group after loss detection, even if a single loss occurred; there was no propagation time (infinite bandwidth); feedback packets were never lost; buffer was infinite for all parties involved. Implementations of the protocols A and N1, when built using Simmcast, are shown below, for the sender side only[5]. The input arguments and symbols were taken from [20]: $E[X_p]$, $E[X_a]$, $E[X_n]$, $E[X_t]$ representing, respectively, the mean times for packet transmission, ACK handling, NACK handling, and retransmission timeout. The values employed in [20] are, respectively 1ms, 0.5ms, 0.5ms, and 0.024ms, for the sender side, and no delay at the receiver side. Consequently, the simulation was configured with $T_{send} = 1$ms and $T_{recv} = 0.5$ms; in case of A sender, timeout overhead was explicitly modelled using `sleep`($E[X_t]$).

[5]the study of receiver capacity is omitted since the sender constitutes the bottleneck.

Protocol A, whose pseudo-code is shown in Figure 3, uses selective retransmission. One packet is transmitted at a time, and retransmitted until at least one ACK has been received from every receiver. Data packets are uniquely identified by a sequence number. Since in [20] there is neither propagation time nor any delay at receivers, the round-trip-time always equals the time to transmit a packet. Hence, the retransmission timeout for A sender is the best possible: the packet transmission time, or T_{send}. In other words, the sender can check if all ACKs have arrived right after finishing with transmission.

```
SENDER:
        while there are packets to be transmitted
           send packet of sequence seq
           while there are acks ready to be received
              receive ack
              add ack to ack list if not already there
              if ackList is complete
                 increment seq
                 clear ackList
              else
                 wait on E[Xt], the timeout handling time
RECEIVER:
        while true
           receive data packet of sequence seq
           send positive acknowledgment for sequence seq
```

Figure 3: A - Ack-based Protocol

For protocol N1 (and many other receiver-initiated protocols), loss detection is carried out at receivers, through a gap in the packet sequencing. For it is not possible to state how many packets in sequence will be lost by any receiver, the sender must send all its packets, instead of going one at a time as done with protocol A above. To have multiple outstanding packets and recovery going on would make the model unnecessarily complex; to simplify it, the implementation of losses is simulated *at the receiver side*. In case of drawing a loss, a NACK packet may have to be transmitted (it is not when another copy of such packet had been already successfully received). The sender transmits a packet, and similarly to protocol A, can immediately afterwards check for any existing NACKs that may be available. Then any existing NACKs are received; one NACK is sufficient to make the sender retransmit the packet. The pseudo-code for the N1 protocol is shown in Figure 4.

The results for the models simulated with Simmcast are shown in Figure 5, along with the ones provided in [20]. The two lines correspond to the analytical results, whereas the bullets represent the simulation ones. The analytical results were obtained by applying formulae (4) and (10) for the values used in [20] (for illustration purposes taking the particular case of loss probability of 0.05). The difference between analytical and simulation results was generally under 1%. As an aside, note that neither A nor N1 protocols scale well: as shown in Figure 5,

```
SENDER:
        while there are packets to be transmitted
           send packet of sequence seq
           while there are nacks ready to be received
              receive nack
           if no nacks have been received
              increment seq, advancing to next packet
RECEIVER:
        while true
           receive data packet of sequence seq
           if draw packet loss
              if packet seq had not been already successfully received
                 send negative acknowledgement regarding packet seq
```

Figure 4: N1 - Multicast Nack-based Protocol

both protocols “break” before a group size of 100 receivers is reached, which is equivalent to 10% of the domain shown.

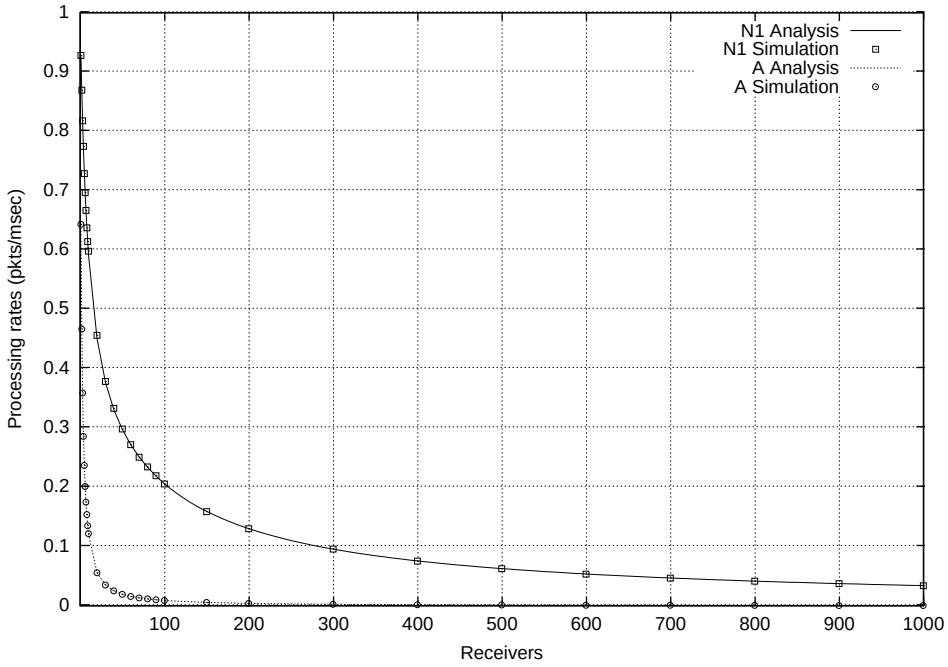

Figure 5: Comparison between analytical results and Simmcast simulations for A and N1.

Note also that using different techniques, like analytical evaluation and simulation, are positive, complimentary parts of the scientific process of studying protocols. Expressing the analytic models and its simplifying assumptions with simulation might appear to point out to the reader limitations of [20]; however, this is not so, since the goal of that study was to determine maximum processing rates (and thus under best conditions) of sender and receiver-initiated protocols, not their typical performance.

4 Related Work

There has been considerable work on network simulation, resulting in several interesting tools. These may be divided into *simulators* and *testbeds.* Testbeds allow certain network conditions to be emulated by filtering out, delaying or duplicating packets. Examples of testbeds are Delayline ([10]), Dummynet ([17]), ComFIRM ([12]), and x-sim ([4]). Delayline is a library that must be linked with a distributed application, so that the latter can be tested in stringent network conditions, where packets are delayed or lost. Dummynet follows the same principle, introducing delays and losses, however acting as an additional protocol layer. It has a compatibility limitation: the software is bound to a specific system and kernel release of FreeBSD; if to be used in a different Unix system, Dummynet would have to be adapted to that specific kernel. Like Dummynet, ComFIRM is a kernel-based tool, which allows *certain* packets to be delayed, duplicated or discarded, according to a set of user-defined rules. ComFIRM is restricted to Linux: it lies within the Linux kernel, and input is given through the `/proc` file-system. Finally, x-sim is a layered network testbed highly integrated on the x-kernel ([16]); at the bottom layer, x-sim employs an emulation layer called SIM. A protocol graph is instated on top of SIM; packets that are sent by the upper layers are acted upon by SIM and then transmitted out or returned to the upper layers. All communication among peers typically occur in a single node. The drawback with x-sim is that it is highly integrated into the x-kernel, requiring: (a) a port of the x-kernel to be available to a given architecture; (b) the x-kernel software to be installed in the desired machines; and (c) users to learn about x-kernel before they can learn x-sim.

The above tools do not allow *all* conditions to be tested, because they are limited by *existing* conditions. For example, if a multicast protocol is to be evaluated with Dummynet, sender and all receivers will run in the same machine, which might overload the system and alter the timely behavior of the protocol; hence, timeouts would occur more frequently. Another limitation is that results cannot be reproduced. By changing existing conditions, the network testbeds modify real-life systems so that protocols can be evaluated/tested in particular scenarios. Because there are many sources of non-determinism in real systems, results of a given experiment will not necessarily result the same every time.

Simulators, in contrast, must give reproduceable results. Besides, simulators allow any known specific condition to be tested; in general, the developer has much more control over the experiment with simulation than with testbeds. However, the model to be simulated represents an *abstract view* of the actual protocol and underlying network (with its topology, connectivity, packet loss correlation, traffic patterns, etc.). Building a simulation model involves making simplifying assumptions to help focus on the relevant aspects of the study. Without such simplifications, the model would be as complex as the system it is meant to be simulating. Nonetheless, the accuracy of the simulation results depends upon how valid the initial assumptions were ([13]).

One of the most popular examples of network simulator is the VINT ns simulator ([5]). Ns is result of a DARPA-funded research project whose aim is to build a network simulator that will allow the study of scale and protocol interaction in the context of current and future network protocols. Ns has been used to support research on the study of TCP performance, network dynamics under multiple protocol interaction, multimedia protocols, queuing policies at routers, reliable multicast protocols, and congestion control. Ns has the advantage of being a resourceful simulator: it supports a large amount of technologies and actual protocols. Unlike Simmcast, ns employs two languages (a dual model), namely C++ and OTcl (one of the existing versions of object-oriented Tcl). Protocols are written in C++, while simulator software is predominantly C++, and simulation scripts are fully OTcl. This permits the script to access parts of the implementation and vice-versa. Being resourceful and dual has the disadvantages of making the code unnecessarily complex: the simulator is over 200,000 lines of code, of which approximately half is written in C++ and half in OTcl; besides, there are places where C++ and OTcl are mingled together. To develop a new protocol, the designer needs to understand the internal workings of ns, modify its Makefiles, and recompile ns with the new protocol. For example, to create new packet types, ns header files must be changed, clearly not a clean approach. In addition to that, unlike Simmcast, ns does not support the *process approach*: hence ns does *not* have support for multi-threaded protocols, nor it can simulate processing delays. A final limitation of ns is that communicating agents must be bound through the OTcl simulation script, which makes hard for a single ns agent to transmit to multiple addresses. All of the above factors have a negative impact on the development of more complex protocols, in particular of reliable multicast protocols.

Simmcast is tailored for multicast protocols, as shown in Section 2, even though it can be used to the design of unicast protocols as well. It supports multi-threading, allowing the specification of protocols that are built of simpler, synchronous components, which can greatly simplify the protocol design. Simmcast employs a single language, Java: experiments are specified in a description file that invokes Java methods part of Simmcast or of the protocol, through *dynamic class loading*, a powerful feature supported by Java. Simmcast is also a framework: new simulated protocols can be rapidly developed by combining building blocks and specializing code already present. Finally, Java was chosen because although it lacks performance in comparison with languages like C or C++, it compensates in portability, ease of use, and standard library support for threads, networking, and graphics. In terms of simulation, this promotes visualization and parallel/distributed simulation.

5 Concluding Remarks

The contribution of this paper is to introduce Simmcast, a discrete-event, process-based simulation framework for designing and evaluating *multicast* protocols. As such, Simmcast has extensive builtin support to multicast communication. Its framework employs a structure of building blocks that allows the creation of experiments with different abstraction levels. Simmcast is easy to use, for the following reasons. First, the object-oriented programming interface is based on a concise set of primitives, and shields the users from the internal specifics of the simulator. Second, users specify a new protocol by combining available building blocks and inserting custom protocol logic, without the need to recompile any Simmcast packages. Finally, new experiments are created by modifying entry parameters in a simulation description file, again without having to recompile any code.

To illustrate the use of Simmcast, a simulated version of a simple reliable multicast protocol was presented. The comparison between results obtained from analytical evaluation of multicast protocols and the corresponding simulation results showed that abstract models can be precisely expressed with Simmcast. Then, such abstract models can be improved to represent increasing levels of detail, providing precious insight about the actual performance of multicast protocols.

Future work includes adding a “native” concept of communication ports to the model, by having more than one *RQ* in a node. Further improvements are the development of a range of well-known multicast protocols, so that the framework is refined and new building blocks are gradually added. These include all sorts of multicast protocols: *multicast routing algorithms* (the logic for routing nodes), *data dissemination reliable multicast protocols*, *multimedia real-time broadcasting* (and QoS), as well as *distributed group-based applications*. Long-term goals include the exploration of parallel and distributed simulation, and the development of a graphical interface, to aid specifying protocols and experiments, as well as visualizing simulation results.

Acknowledgments

The authors would like to thank the useful comments provided by reviewers, as well as FAPERGS for partly funding the work reported in this paper. The authors also thank Dr. Mark C. Little, for developing JavaSim while at at Newcastle University.

References

[1] B. Bal, “Design and Implementation of a Reliable Group Communication Toolkit for Java”, University of Cornell, available at http://www.cs.cornell.edu/Info/Projects/JavaGroupsNew/papers/Coots.ps.gz, Jan. 2001.

[2] M. Barcellos & P. Ezhilchelvan, “A Reliable Multicast Protocol using Polling for Scaleability”. In IEEE INFOCOM’98, San Francisco, 29 March-2nd. April 1998.

[3] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard, “Simula begin”, Studentlitteratur, Lund, Sweden, 1973.

[4] L. S. Brakmo and L. L. Peterson, “Experiences with network simulation”, In ACM Sigmetrics’96, pp. 80-90, May 1996.

[5] L. Breslau et alli, “Advances in Network Simulation”, IEEE Computer, v.33, n.5, pp. 59-67, May 2000.

[6] J. Cowie, D. M. Nicol and A. T. Ogielski, “Modeling the Global Internet”, Computing in Science & Engineering, v. 1, n. 1, pp. 42-50, Jan 1999.

[7] K. Fall and K. Varadhan, “The ns Manual”, The VINT Project, UC Berkeley, LBL, USC/ISI, Xerox PARC, http://www.isi.edu/nsnam/ns/ns-documentation.html

[8] H. Hüni, R. Johnson, and R. Engel. “A framework for network protocol software”. In Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA’95), pp. 358-369, Austin, TX, September 1995.

[9] F. Howell and R. McNab, "simjava: a discrete event simulation package for Java with applications in computer systems modelling", In proc. First International Conference on Web-based Modelling and Simulation, San Diego CA, Society for Computer Simulation, Jan 1998.

[10] D. B. Ingham and G. D. Parrington, "Delayline: A Wide-Area Network Emulation Tool," Computing Systems, v.7, n. 3, pp.313-332, 1994.

[11] B. Krupczak, K. Calvert, M. Ammar, "Implementing Protocols in Java: The Price of Portability", In IEEE INFOCOM’98, San Francisco, 29 March-2nd. April 1998.

[12] F. O. Leite, “ComFIRM - Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional”, M.Sc. Dissertation, CPGCC-UFRGS, Dec. 2000

[13] M. C. Little, “JavaSim Users Guide”, Public Release 0.3, Version 1.0, http://javasim.ncl.ac.uk

[14] D. Madhava Rao, R. Radhakrishnan, and P. A. Wilsey, “FWNS: A Framework for Web-based Network Simulation”, In Proc. of International Conference On Web-Based Modelling & Simulation, WEBSIM’99, Volume 31, Number 3, 9-14, January 1999.

[15] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya, “Reliable Multicast Transport Protocol (RMTP)”, IEEE Journal of Selected Areas in Communications, v.15, n.3, pp.407-421, 1997.

[16] L. Peterson, B. Davie, and A. Bavier, “*x*-kernel Tutorial”, The University of Arizona, Computer Science Department, Jan. 1996, http://www.cs.arizona.edu/classes/cs525/tutorial/tutorial.html

[17] L. Rizzo, “Dummynet: a simple approach to the evaluation of network protocols”, ACM Computer Communication Review, v. 27, n.1, Jan. 1997.

[18] M. A. Rodrigues, S. Colcher, and L. F. Soares, “Um Framework para a Provisão de Serviço de Multicast em Ambientes Genéricos de Processamento e Comunicação”, Em SBRC’99 - XVII Simpósio Brasileiro de Redes de Computadores, p.206-221, SBC 1999.

[19] D. C. Schmidt, “The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software”, In 12th Annual Sun Users Group Conference, pp.214-225, San Francisco, CA, 1994. (see also http://www.cs.wustl.edu/~schmidt/ACE.html)

[20] D. Towsley, J. Kurose, and S. Pingali, “A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols”, IEEE Journal of Selected Areas in Communications, v.15, n.3, pp.398-406, 1997.

[21] B. Whetten et alli, “Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer”, RFC3048, IETF, Draft 3, January 2001, ftp://ftp.rfc-editor.org/in-notes/rfc3048.txt