Simulating Group Communication Protocols Through an Object-Oriented Framework*

Hisham H. Muhammad  Marinho P. Barcellos

PIPCA- Programa Interdisciplinar 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
{hisham,marinho}@exatas.unisinos.br

Abstract
This paper discusses the design and implementation of Simmcast, an object-oriented framework for network simulation with specific support for group communication. The design of the framework is focused towards simplicity and extensibility. The aim is to allow a spectrum of experiments ranging from evaluation of abstract group communication models to simulation of more detailed multicast protocol behavior. Simmcast employs a process-based discrete-event model on which building blocks are combined and extended in order to create new simulation environments. Network parameters are given in terms of numeric distributions (fixed or probabilistic), which can be replaced without recompilation. This extensive use of numeric distributions combined to the extendible framework structure allow an abstract experiment to evolve into a detailed one by progressively increasing the level of detail and sources of non-determinism of the constituting blocks.

Keywords: network simulation, framework, group communication

1 Introduction

There are many classes of applications addressed by multicast protocols, such as bulk-data dissemination, live broadcasting of multimedia contents over the Internet, multimedia video-conferencing, as well as dependable distributed applications based on group communication1. All these applications are based on some kind of multicast protocol as the underlying abstraction, which include hierarchical receiver-initiated reliable multicast protocols, multicast layering protocols, and many-to-many group communication protocols that provide message ordering schemes, delivery guarantees, and consistent group membership. Each class possesses peculiar characteristics that should be evaluated accordingly, usually in function of protocol input parameters (e.g., buffer size, packet size, number of senders/receivers) or changes in specific network conditions (e.g., error rates, congestion).

Three well-known techniques have been extensively used to evaluate the performance of communication protocols and distributed systems: analytical evaluation, experimentation (measuring a real system) and simulation. Examples of multicast protocol evaluation using these schemes are [17], [13] and [9], respectively. Out of these techniques, simulation offers an advantage in that it can be used not only to evaluate protocol performance according to given metrics, but also to better understand protocol interactions and identify potential protocol pathologies. Further, simulation allows one to experiment with a protocol under dynamic scenarios, such as scheduling temporary link and node crashes.

Simulation involves a modeling process that evolves into later execution on a simulation tool. There are many sensitive aspects that will rule the choice of such tool, and these are mainly affected by the characteristics of the model. The most commonly chosen approaches are: using a ready-made simulator which will hopefully fit the project’s needs, or develop a new one, specifically designed for the intended experiments.

Both approaches present clear disadvantages: a large, monolithic simulator would limit the flexibility of the researcher, forcing him/her to model the problem in question into the environment provided by the simulator itself. While some excessively detailed parts of this environment will appear as overkill to the problem, others will not provide all of the required functionality. On the other hand, the complete development of a dedicated simulation tool from scratch is not practical, since the amount of resources dispensed in such a task would detract the researcher’s focus from the project.

Therefore, a different approach is needed, combining the best of both worlds. Such approach would imply having a simulation toolset that relieves the researcher from dealing

---

1 the terms ’multicast protocol’ and ’group communication protocol’ are used interchangeably in this text to refer to protocols whereby the same message needs to be efficiently transmitted to a number of receivers, which are organized as a group. The ’group membership’ represents the composition of a group in a given instant in time.
with generic tools (thus tailored to the domain of interest) and constructing an entire simulator, but at the same time allows him/her to mold the aspects of the simulation environment, knowing that it is bound to change as the experiments evolve.

Face to the above requirements, a simulation architecture was proposed and conceived as an object-oriented framework ([8]), called Simmcast. It allows a multicast protocol or group communication application to be investigated by expressing it as a custom simulation model built on top of provided, abstract, building blocks, that are linked together through a process-based discrete-event simulation engine.

Simmcast is based on Java and JavaSim ([10]), a simulation toolkit whose aim is to emulate the Simula language ([11]) facilities in Java. JavaSim, and thus Simmcast, follows the same process-based, object-oriented, discrete event model introduced by Simula in the late 60s. Hence, Simmcast uses a familiar and proven simulation model, used in a wide range of applications, including group communication protocols.

The paper is organized as follows: Section 2 describes the architecture from an architectural point of view and how this architecture was translated into the interface of Simmcast. While Section 3 discusses the implementation, Section 4 compares with related work, and Section 5 concludes the paper.

2 Architecture Description

The entities of the simulation and the relations between them must be represented throughout the architecture in a consistent way. The proposed architecture does so by defining a framework where extendible building blocks are combined in order to describe the simulated network environment, and on top of it, the multicast protocol under investigation. In this section, an overview of the framework structure is presented (Section 2.1), followed by a more detailed description of the abstract building blocks and the interactions between them (Section 2.2). Other aspects relevant to the architecture, such as the tracing interface and experiment configuration are also discussed (Sections 2.3 and 2.4, respectively).

2.1 Framework

The use of frameworks increases software reusability, which can lead to advantages like reduced development effort and more robust code through multiple reuse and refinement of the framework ([8]). Besides, a framework is also advantageous because the architecture is not intended to a particular type of multicast protocol or application, and thus needs to employ an abstract model that can be specialized according to the needs of the user. Frameworks seem particularly appropriate for simulation, since substantial part of the code can be reused between simulation experiments. Indeed, frameworks have been applied to network simulation (e.g., [4], [12]) and to the development of network protocols (e.g., [6], [15]).

In Simmcast, two main advantages stem from the use of a framework. First, the simulator internals are greatly simplified since they are constructed defining what is called the framework’s kernel: this is a minimal, very abstract layer built around the simulation engine of JavaSim, providing basic packet-level communication and group management. Second, 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. One builds a custom environment, but still uses most of the framework functionality without having to reinvent that functionality.

The simulator interface is defined as an API with the typical communication and timer operations, as well as a suitable (concurrent) software architecture to help designing simulated group communication protocols and applications. This API is not unlike Java’s own networking or threading interfaces (in fact, the thread mechanisms of JavaSim are subclasses of the Thread class from Java). Therefore, the primitives of the simulator are no more intrusive in the designer’s code than actual system calls would be; it is usually even less intrusive, since simulated code usually deals with a lesser number of exception cases. Simmcast offers a series of methods which define a concise set of primitives, as listed in Table 1, through which the user will dictate the interaction between building blocks. The idea is to have the simulator primitives inserted into the user’s code, and not the other way around. Simmcast also offers a tracing mechanism that allows custom accounting to be dynamically connected at run-time (see Section 3.4).

<table>
<thead>
<tr>
<th>Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>send a packet to a given destination</td>
</tr>
<tr>
<td>receive</td>
<td>blocks until a packet is received</td>
</tr>
<tr>
<td>tryReceive</td>
<td>attempts receiving a packet or returns “null”</td>
</tr>
<tr>
<td>join</td>
<td>receiver joins a given multicast group</td>
</tr>
<tr>
<td>leave</td>
<td>receiver leaves a given multicast group</td>
</tr>
<tr>
<td>setTimer</td>
<td>configure timer to expire in a given time</td>
</tr>
<tr>
<td>cancelTimer</td>
<td>cancel an existing timer</td>
</tr>
<tr>
<td>onTimer</td>
<td>method invoked when a timer expires</td>
</tr>
<tr>
<td>sleep</td>
<td>put the current thread to sleep for a given time</td>
</tr>
<tr>
<td>wakeUp</td>
<td>wake up another thread that may be sleeping</td>
</tr>
</tbody>
</table>

Table 1: List of Primitives.

2.2 Building blocks

Building blocks have been used before to design actual multicast protocols ([18]), and we see them as appropriate for simulating multicast/group communication protocols in general. Building blocks are the key to the modularity of the
simulation, as they serve two purposes. First, they are specialized by providing additional code through inheritance, defining protocol logic or other specific behavior. Also, through composition, experiments are described as a combination of a set of building blocks. We identified a set of basic elements needed to represent a network simulation which led to the following building blocks: node, thread, path, group, network, packet, and stream. For each building block, there is a corresponding class in Simmcast. Below, the building blocks are presented.

Nodes are the fundamental interacting entities, and uniquely identified by an integer. Their correspondence in the model is not dictated by the simulator: 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. Threads simplify a protocol because they allow the developer to model a concurrent architecture using a set of simpler entities that behave synchronously, even though this bears a price on performance and scalability.

Nodes will also contain at least three queues, one to send out packets, another to receive in packets, and a third to record asynchronous events, as follows. Let the set of existing nodes in a simulation be represented by \( x_1, x_2, \ldots, x_N \), where \( N \) is the total number of nodes. A node \( x_i \) will have a sending queue \( sg_{i,j} \) for any node \( x_j \) that \( x_i \) is connected to (depending on the topology defined), and also a single receiving queue \( rq_i \), to which all arriving packets will be added (from any of the paths that arrive at \( x_j \)). The capacity of \( sg \) and \( rq \) can be explicitly set or left unlimited. The \( sg_{i,j} \) queue is served according to bandwidth to \( x_j \), while \( rq_i \) is served according rate in which reception operations (in Simmcast, receive() and tryReceive() ) are invoked. Adding a packet to \( sg \) with send() takes \( t_{send} \) time, and this value can be used to limit the sending rate by the protocol; taking a packet from \( sg \) with a successful reception takes \( t_{recv} \) time, and this value can be used to limit packet receiving capacity.

Finally, a node \( x_i \) will have a timer queue, represented as \( tq_i \), to record future asynchronous events. 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 can allow a node to suspect that a network partition has occurred. Timers can also be used to implement periodic processing behavior, like processing timers only at every 200 and 500ms (such as TCP). Timed events can be controlled in Simmcast by simply overriding the onTimer method of the Node class.

The Node and NodeThread classes are identified as hotspots in the framework. Several methods offer points of extensibility, be them abstract (such as execute() in the NodeThread class) or empty (such as begin() in the Node class). Being key factors in this first moment during the development of a simulation experiment, these building blocks have a white-box characteristic, allowing that even the most basic primitives from Simmcast, such as send() to be extended through inheritance.

Nodes are connected by paths. Paths represent a packet flow between two nodes, and thus their meaning in the model depends on what the nodes themselves are representing. In other words, the concept of node and path are dissociated from a physical connotation. A node can represent a module or a layer in a protocol graph, as much as it could represent a router or an application agent. Paths are used to connect these nodes, and could represent a packet queue, a physical link or a logical path between two end-nodes. All paths are unidirectional: to model a bidirectional physical link two Simmcast paths must be used. A path from \( x_j \) to \( x_i \) is represented by a path, or \( pq_{i,j} \), that holds packets in transit from \( x_j \) to \( x_i \). A path will accept the following properties: bandwidth, packet loss probability, and propagation delay. The bandwidth (along with packet size) will determine the service rate of the sending queue (also the admission rate into \( pq \), the path queue). The propagation delay will determine the time a packet spends in \( pq \). Loss probability is applied when the packet arrives at the destination \( rq \). Note that if \( x_i \) and \( x_j \) are two consecutive layers of a protocol, then bandwidth should be infinite and the delay zero, so that packets go straight from \( sg_{i,j} \) to \( rq_j \); in this case, packet loss probability equals zero.

The status of the \( pq \) queues is returned to the user only inside the trace mechanism (the TraceGenerator class), for accounting purposes. All classes that implement paths are black-box. The management and resolution of paths is done at the kernel of the framework; the user’s code cannot refer to paths explicitly, as the transmission and reception primitives make use of the nodes’ integer identifiers as the only means to refer to them. The creation of paths itself is controlled: the Node class implements a Factory design pattern ((51)) for this end, interfaced through the addPath() primitive, to be used dynamically in the simulation description file.

The concept of group is paramount to multicast protocols, so defining group as a primitive building block gives it a “first-class” status in the simulation environments designed using the framework, and allows abstract simulations of conceptual models of group communication to be performed without concerns on unrelated, low-level issues. Simmcast allows a group to be described either statically, through the experiment description file (see Section 2.4), or dynamically, through join() and leave() primitives (as in Table 1) executed by the protocol nodes. These operations are performed instantaneously. For example, if a node \( x_i \) is member of group \( g \), and \( x_i \) requests to leave \( g \), from then on any transmission to \( g \) will not deliver the packet to \( x_i \). However, if a

---

2In fact, a node may even represent a series of hosts. The netNode in the "utilities" package (simmcast.util) does that, simulating a "network cloud".

3Since the simulation environment is modeled by the designer, the user-provided code may also reference the nodes directly.
A network is an arbitrary combination of nodes, and all paths connecting them. No specific routing scheme is enforced, so that different kinds of routing schemes can be used interchangeably. Therefore, for a packet to be transmitted from $x_i$ to $x_j$, a path must have been created connecting these two nodes (that is, $\exists p_{ij}$). Likewise, for $x_i$ to send a packet to a given group $g$, it must be directly connected to each of the elements $x_j$ of $g$ (for every $x_j \in g, \exists p_{ij}$).

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. Inheritance can be used to easily define packet types for new protocols. Nonetheless, a packet is constructed as a container of an arbitrary object, and this object may be defined to be the desired new packet type.

![Packet flow through packet queues and their service times, for an example of multicast transmission from a node $x_i$ to nodes $x_j$ and $x_k$.](image)

Figure 1: Packet flow through packet queues and their service times, for an example of multicast transmission from a node $x_i$ to nodes $x_j$ and $x_k$. Each parameter of a simulation model can be fixed or of random nature. In Simmcast, they are read from what we termed number stream objects. These are created according to user specified parameters in order to generate a sequence of values. In the case of a fixed parameter, the number stream will always return the same configured value. Otherwise, in every access a new number will be drawn from the pre-configured random distribution stream (either uniform, normal, exponential, hyperexponential, Erlang, or user-defined). The characteristics of the random distribution (such as mean, standard deviation or upper and lower bounds), can be provided at stream creation time. This way random latency values can be assigned to logical paths, mean processing times, etc. A given number stream can be associated to an individual parameter (e.g., end-to-end latency of a given path), or else can be shared (e.g., to implement a global packet loss probability).

Number streams are important because they allow the user to transparently replace fixed values by random ones in the simulation model. In line with Simmcast philosophy, this can be employed to study a simpler model, and once understood, gradually add sources of non-determinism to it, while assessing the impact of such changes.

Finally, there is a utility package, made of derived building blocks (blocks which are extensions of the primitive building blocks presented in this section) and other contributed convenience objects that are frequently used in protocol design, such as AckWindow and SlidingWindow.

2.3 Simulation output

Several different metrics are used in network simulation, depending on the nature of the experiment. Some examples are total required bandwidth, average time to packet recovery at receiver nodes, number of "late" packets, total number of packets exchanged, amount of dropped packets, and time until all group members reach agreement on membership after a failure of given group member. Translated into the Simmcast architecture, these metrics can be generalized into different forms of accounting a small group of event categories, which we set apart as traceable events.

Traceable events are considered to be inclusion and/or removal of an element to/from its queues ($sq$, $pq$, $rq$, and $tq$). For example, according to Figure 1 (in Section 2.2), the following events will take place when a packet is transmitted between two nodes (assuming it is not lost): adding a packet to $sq$, moving the packet from $sq$ to $pq$, moving the packet from $pq$ to $rq$, and removing the packet from $rq$. Scheduling a future asynchronous event is represented by enqueuing objects to $tq$, which are removed either automatically, representing timer expiration and subsequent event triggering, or explicitly, representing timer cancellation.

Simmcast provides a unified output interface to where all events in the simulation are reported, the TraceGenerator class. TraceGenerator has a read-only view of the entire simulation scenario. Reported information includes the event time and basic information about every packet as well as the current internal state of all queues. Some of this information is not available inside the simulation environment itself, in order to maintain the consistency of the queue model. In this sense, TraceGenerator can be viewed
as a protected sandbox where queue information can be manipulated without possibly compromising the execution (as opposed to, say, inserting counters inside the simulator or user code). Simmcast has a SimmcastTraceGenerator subclass, that generates native traces using this interface. The user may also subclass TraceGenerator, in order to perform any desired accounting on this data. Simmcast makes available, through JavaSim, statistical classes to help consolidating output information. During the simulation, values are input to this class (like with a standard Java input stream), and at any time partial values can be displayed. Examples of classes are Mean, Variance, Histogram and TimeHistogram.

2.4 Instantiation and execution

A simulation experiment using Simmcast is done in two stages. First, it is necessary to instantiate the framework, constructing the simulated protocol or application model. A new protocol can be built by combining existing building blocks and specializing Node and related classes. Depending on what kind of multicast communication is simulated, there may be one kind of agent (e.g., peer agent or replica manager in a group application), two kinds (e.g., sender/receiver, or client/server, or master/slave), or more (e.g., sender/proxy/receiver).

The second stage is to execute the resulting model passing to it a series of parameters. These parameters will list the nodes and define their connectivity, thus specifying the network topology. Different levels of network representation are possible, but the network is typically either a partially connected graph with diameter 1, or a given connected graph with arbitrary diameter (but not more than \( N - 1 \)). The abstract model of Simmcast does not include routing, only direct connections among nodes. Hence, in the latter case some nodes (internal to the network) will have to include routing logic, or a multicast tree topology will have height equals two. A higher level of detail can be easily obtained by making part of the nodes act like simple routers. This is an important design decision upon the simulation model, following the Localized Cost principle ([11]), choosing not to impose unnecessary complexity / processing cost. All these settings can be defined in a simulation description file, which is simply a text file with a series of constructor and method calls to be performed by Simmcast, as seen in Figure 2. We decided to use the class and method reflection features in favor of linking a complete scripting language. Indeed, only a few commodities such as macros and array notation are allowed, while other typical programming language features such as looping constructs were considered but discarded to avoid spreading the experiment logic through different languages and environments. This way, besides maintaining the simplicity of the system, a better “separation of concerns” is guaranteed, constraining the use of the description file mostly to specifying the topology and startup parameters.

### Figure 2: The simulation description file of a client/server group architecture.

3 Implementation

The use of an object-oriented framework model and a high level of abstraction contributed to a good decoupling between the interface that is presented to the user (discussed in the previous section) and its implementation, whose structure often differs considerably. In this section, we discuss how the architecture of Simmcast was translated into its implementation.

3.1 Kernel structure

At the lowest level of the framework lies JavaSim, which is a process-based, discrete-event simulation engine. Its basic entities are SimulationProcess objects, each corresponding to a single thread of execution. The Simmcast model, on its turn, is based on Node objects. A Node object serves as a multi-threaded entity; in practice, it is a container for multiple NodeThread objects (NodeThread is a subclass of SimulationProcess). Additionally, there is a single object of the Network class, that holds the Node objects and serves as a centralized entity controlling the simulation.

In the user level, there are two types of possible communication: intra-node (i.e., between threads of the same node), and inter-node (between nodes). Communication between nodes can be performed directly through their common Node object. Internally, no synchronization control needs to be performed as the process model corresponds to cooperative multitasking. Communication between nodes is done transferring packets using the primitives of the API (send(), receive(), tryReceive()). The implementation of this type of communication is described below.
3.2 Communication between nodes

Each Node, by default, possesses one thread to control the flow of events: the EventScheduler. This is an internal structure, invisible to the user’s code. Every EventScheduler has an EventQueue associated to it. Packet transfers are implemented as “packet departure” and “arrival” events in the EventQueue objects of the sender and receiver, respectively. There is no \( pq \) per se in the implementation: it exists only conceptually (as discussed in Section 2.1).

When a Node object issues a send() operation, the node verifies, among the paths connected to it, which one leads to the receiver node. The packet is then added to the sender’s \( sq \) (which is held in the Path object) and a “packet departure” event (represented by a DepartureEventItem object) is scheduled in the EventScheduler of the sender node. When this event is triggered (after \( Tsnd \) time), the packet is removed from the \( sq \) and encapsulated in an ArrivalEventItem object. This event is scheduled in the EventScheduler of the receiver node, for an amount of time that will correspond, conceptually, to the time the packet spends in \( pq \). When this event is triggered, the packet will be decapsulated and added to the receiver’s \( rq \), becoming available to reception operations. A delay of \( Trcv \) time will be forced onto the Node whenever a successful receive() or tryReceive() is completed.

3.3 Asynchronous events

Many network protocols are based on the concept of time-outs (see Section 2.2). Asynchronous events such as time-outs can be simulated using the thread facilities provided by Simmcast, but these would appear only implicitly in the resulting code, as a mixture of sleep() and wakeUp() calls. We chose to express asynchrony events explicitly in our conceptual model as manipulations of a timer queue \( \{ q \} \), and provide a specific API for this category of events (setTimer(), onTimer(), and cancelTimer()).

This API for asynchronous events is implemented over the same event infrastructure used by packet transfers. The setTimer() method will add an UserEventItem object to the EventQueue found inside node’s EventScheduler, using an arbitrary, user-defined object as an identifier. Again, there is no \( q \) in the implementation. If no cancelTimer() call is issued to remove the event from the queue, then it will be eventually triggered, resulting in a call to onTimer(), a method defined by the user, which will receive the object set as the event identifier and finally handle the event.

In most timeout-based protocols, user-defined asynchronous events are generated at a high rate, and usually the majority of them is canceled before being triggered. Managing these events in the node’s own event queue optimizes the traversal when searching for them and reduces the load on the main event queue of the simulation engine: if the event is canceled before reaching the the head of the node’s private event queue, it is never fed to the simulation engine, and no JavaSim events are generated at all.

3.4 Tracing mechanism

Tracing is a fundamental part in simulation, but one that is usually overlooked in the design of network simulation tools, since most of them produce outputs similar to those given by familiar network monitoring facilities (like tcpdump) and do not explore the full possibilities offered by simulation. Also, the practice of embedding logging and debugging controls throughout the code can compromise its clarity and understandability, and often offers little room for customization.

We wanted the tracing system of Simmcast to be, conceptually, very high level. Further, it should be dynamically pluggable into the simulation environment and yet give full read access into the experiment’s status, allowing custom trace routines to be developed by the user. This was achieved through a design based on a series of hooks at the lowest levels (“below”user code) that monitor every packet transfer or loss, and changes to nodes and paths. These hooks are connected, through the Network object, to a separate subsystem (the TraceGenerator), following an Observer design pattern ([5]). TraceGenerator is an abstract class, which different subclasses may inherit from for specific purposes: an application of the Strategy pattern.

As an additional advantage, the above scheme allows conditional tracing, based on the current state of the simulation. It can be used, for example, to count the number of “ACK implosions” derived from lack of scalability in a reliable multicast transmission. Instead of post-processing trace files to identify these occurrences, this was done in the program itself, but still without blending with the experiment code.

Note that the simulation status that appears to the tracing mechanism is analogous to the conceptual model from Section 2.2, not to the actual implementation (i.e., in terms of \( sq, rq, pq \) and \( t_q \) not in terms of the event queues).

4 Related Work

There has been considerable work on network simulation, resulting in several interesting tools. In this section, we comment on some design approaches taken by other simulation projects in the area of network protocols. Many tools allow certain network conditions to be emulated for higher layers, by delaying, duplicating or filtering out packets as they are transmitted or received by one or more communicating hosts in an actual network. The main advantages of this technique are that it provides more realistic results and works along with real protocols and applications. On the other hand, it
does not allow arbitrary conditions to be tested, since the simulation is restricted and, more importantly, affected by, the environment in which it is run (e.g., can increase latency but not reduce it). Another limitation is that results are not always reproducible, because of the many sources of non-determinism present in the real systems the tool has to interact with. Hence, these tools have a more specific applicability. Examples that follow this approach are Delayline ([7]), Dummynet ([14]) and x-sim ([2]).

In contrast, general-purpose network simulators attempt at giving reproducible results and having a wider applicability. In a general-purpose network simulator, a model of an entirely simulated network environment is presented. The protocol researcher will design a topology using the different elements available in the given model and extend the latter adding the code of the protocol under investigation.

One of the most popular examples of network simulator is VINT ns ([3]), 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 originally used to support research on the study of TCP performance, and has then extended in various areas, such as network dynamics under multiple protocol interaction, multicast protocols, real-time protocols, and congestion control. ns has the advantage of being a resourceful simulator: it supports a large amount of technologies and actual protocols. From a simulation standpoint, ns has the main disadvantage that it does not support the process-based model: hence it does not have support for multi-threaded protocol design, nor it can simulate properly processing delays.

Another general-purpose simulator is IRLSim ([16]), developed using the Parsec simulation language. Originally developed specifically for RSVP (a resource reservation protocol), it has since been extended to support transport protocols. IRLSim employs a modular structure, even though this structure is dictated by a number of simulation layers added to the original RSVP layer. The level of abstraction used by IRLSim is fixed, and very low. While this can be useful for transferring real code to and from the simulator, it may inflict its applicability for high level simulations where the detail in the lowest network layers could be simplified.

Simmcast approaches network simulation from a different perspective. While in general-purpose network simulators the new protocol is added to the existing model, in Simmcast the model is built along with the protocol, through instantiation of the framework. This bring clear advantages in terms of flexibility. Simmcast can be thought of as a network-research-oriented simulation engine with built-in support for group communication/distributed systems facilities. Still, it is more than that. Its framework structure, while maintaining the designer’s modeling freedom, conserves the architecture’s sense of hierarchy, by keeping entity constitution, composition between entities, protocol code and accounting separated.

5 Concluding Remarks

This paper described the design and implementation of Simmcast, a network simulation framework oriented towards multicast/group communication protocols. The application of a simulation framework stands between using an existing simulator and creating a new one. Simmcast is, therefore, similar to using a network simulator in the sense that it provides the basic mechanisms required for performing a protocol experiment, in the form of an API of communication primitives. At the same time, it resembles a simulation engine, as the user builds the simulation environment the simulation will run on, however, not from scratch, but from a set of building blocks that are designed to a specific domain of interest.

Control over the level of abstraction employed was one of the main goals in the design of Simmcast. The correspondence between a building block and its meaning in the simulation environment is up to the user. The definition of group as a fundamental building block allows not only a cleaner manipulation of group information in the simulation code, but also their representation in the same abstraction level on which the environment itself is based. One creates a model of nodes, groups and paths on which the protocol runs; not a model of nodes and paths over which groups are managed and then the protocol runs. This allows, for instance, the evaluation of conceptual models of group communication through simulation.

We presented how the framework was implemented and how queue interactions were represented in terms of simulation events, while retaining, at the interface level, a process-based (multi-threaded) view of the simulation. We also discussed design and implementation aspects from some of the facilities of Simmcast. The TraceGenerator is a safe and high-level interface for accounting. The simulation description file is a mechanism that, through use of the reflection API from Java, allows a flexible definition of topology and modification of simulation parameters (either built-in or user-defined).

We found the object-oriented framework model to be very well suited for the development of a network simulation tool. Its flexibility and modularity allowed the design to evolve and mature to this point. Applications of Simmcast are actively in use by our research group and others\(^5\), and in the future we expect to expand its utility package with more derived building blocks. Long-term goals include the development of a graphical interface, to aid specifying protocols and experiments, as well as visualizing simulation results, since no existing visualization tool seems to cope with the dynamic characteristics of link behavior in Simmcast.

\(^5\)Simmcast is free software, distributed under the GNU General Public License, and is available on the Internet at http://inf.unisinos.br/~simmcast.
Acknowledgments

The authors would like to thank Dr. Mark C. Little, for developing C++SIM and JavaSim while at Newcastle University. Thanks also to André Detsch and Guilherme Bedin for their contribution in earlier parts of this work.

References