Journal of Network and Computer Applications 35 (2012) 328–339

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Beyond network simulators: Fostering novel distributed applications and protocols through extendible design

Marinho P. Barcellos *, Rodolfo S. Antunes, Hisham H. Muhammad, Ruthiano S. Munaretti

Institute of Informatics, Federal University of Rio Grande do Sul (INF/UFRGS), Av. Bento Gonçalves, 9500 – Bloco 4, Porto Alegre/RS, Brazil

ARTICLE INFO

Article history:
Received 4 April 2011
Received in revised form
1 July 2011
Accepted 30 July 2011
Available online 6 August 2011

Keywords:
Distributed systems
Simulation
Software engineering

ABSTRACT

Simulation has been of paramount importance to the development of novel Internet protocols. Such an approach typically focuses on one of three domains: wireless and other link-layer technologies, routing protocols, and transport-layer mechanisms and protocols. Existing techniques can tackle well simulation at layers 2, 3 and 4 of the TCP/IP architecture, but are not flexible enough to appropriately deal with application-layer protocols. These require simulators that support the modeling of networks and components with different levels of abstraction. Simmcast is an object-oriented framework that focuses on the necessary flexibility for application-layer protocol research. A simulation can be developed by the simple extension of building blocks that closely resemble components of a real network such as hosts, links and routers. The internal complexity of these components, however, is hidden from the user, so he/she can focus on the implementation of the desired protocol characteristics. This paper describes the flexible simulation architecture proposed and instantiated through Simmcast, and draws lessons from our experience in designing, implementing and deploying it. We also present framework instances used to evaluate application-layer protocols, exemplifying how different kinds of simulations can be developed with Simmcast.

1. Introduction

Protocols are the fundamental entity of computer networks and distributed systems, and their characteristics vary widely according to the purpose and the network layer they belong to. In developing and evaluating protocols, one should consider the peculiar characteristics of the corresponding class being studied. These can be mapped into a range of protocol input parameters or specific network conditions to be assumed or evaluated.

Three well-known techniques have been extensively used to evaluate the performance of network/transport protocols and distributed systems: *analytical evaluation*, *experimentation* (measuring a real system) and *simulation*. 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* interactions and identify potential anomalies. Further, simulation allows one to easily 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 a 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. Unfortunately, this has been all too common, with a proliferation of simulators.

Therefore, a different approach is needed, combining the best of both worlds. Such an approach would imply having a simulation toolset that relieves the researcher from dealing with generic tools and from constructing an entire simulator, but at the same time allows him/her to mold the aspects of the simulation environment as the experiments evolve.

Given the above requirements, Muhammad and Barcellos (2002) proposed a *simulation architecture*, conceived as an object-oriented *framework* called Simmcast. It allows a protocol

* Corresponding author.
E-mail address: marinho@acm.org (M.P. Barcellos).

1084-8045/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2011.07.015

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. Aspects of the simulation environment can be molded and adapted as the experiment evolves, relieving the researcher from dealing with generic tools as well as from constructing an entire simulator. Simmcast follows the same process-based, object-oriented, discrete-event model introduced by Simula language (Nygaard and Dahl, 1978) in the late 1960s. Hence, it employs a familiar and proven simulation model, used in a wide range of applications. The flexibility provided by the framework allows a wide range of protocols, particularly application-level ones, to be properly modeled and evaluated.

After its conception, Simmcast was employed in the development of simulations for studies of application-layer protocols. These simulations allowed us to rapidly acquire results such as, but not restricted to: better understanding of protocol interactions; ideal parameter values to be later used in live experiments; and experience in protocol development and operation in a strictly controlled environment. The use of Simmcast allowed faster acquisition of these results, by greatly reducing the development time necessary to get simulations running. Later comparisons of our simulation results with real protocol implementations in controlled environments showed a very close similarity, demonstrating the potential accuracy achieved by simulators created with Simmcast.

In this paper we aim to present an overview of the development efforts involving Simmcast, including framework design and implementation issues, performance evaluations, and examples of studies which applied the framework. Section 2 categorizes the work on protocol simulation and compares our proposal to similar efforts. The architecture of Simmcast is presented in Section 3, while implementation aspects are addressed in Section 4. This implementation was used in a performance evaluation, and Section 5 provides a performance analysis with some lower bounds on simulation time and space. Studies that employ Simmcast for simulation development are presented in Section 6. The paper closes with concluding remarks in Section 7.

2. Related work

The field of discrete-time simulation is broad, and many tools and techniques have been proposed in the past 10 years. So, if there are too many languages and simulators already, one might ask why devising yet another one. Simmcast is neither a new language nor a simulator application. Instead, it is a *framework* that can be extended into a simulator, unlike similar approaches.

Initial work on network simulation has focused on layers 2, 3 and 4 of the Internet architecture. Link-layer simulators like the one proposed by Lacage and Henderson (2006) focus in the investigation of wireless protocols. NS-2 Developers (2009) were widely used for investigation of routing algorithms at the network layer, or congestion control mechanisms and reliable multicast protocols at the transport-layer. Later simulators also focused on application-level protocols, such as PeerSim (Jelasity et al., 2009) or PlanetSim (Pujol-Ahulló et al., 2009). Below, we comment on the most important initiatives.

VINT NS-2 (Bajaj et al., 1999) is probably the most popular packet-level network simulator employed by the research community. The software was developed combining C++ and OTcl, since it was easier to create more abstract objects in OTcl for configuration purposes while the core of the simulation would be executed using C++, providing good performance where necessary. NS-2 wide adoption by the research community reflects on a large number of protocols from layers 2-4 contributed and incorporated into the simulator.

Early in its evolution, however, NS-2 became too complex. Bajaj et al. (1999) mention the absence of isolation among modules, causing a modification in the source code to generate side-effects and compromise apparently unrelated simulations. The learning curve of NS-2 is steep, possibly leading to misunderstandings of the simulator and incorrect interpretation of results. These limitations led to the creation of NS-3 Developers (2010), which represents a complete rewrite of the simulator. The NS-3 project has a broad set of goals (Henderson et al., 2006), including scalability, extensibility, modularity, emulation, and clarity of design. Like NS-2, NS-3 focuses on the study and analysis of layers 2–4 of the present network model.

Cowie et al. (1999) and Nicol et al. (2003) describe the Scalable Simulation Framework (SSF), a simulation Application Programming Interface (API) with focus on large scale networks, with bindings for both C++ and Java. Its major contribution is to explore the locality of subnetworks to increase the parallelism. The latest official Java implementation of the API, named SSFNet, is available at SSF Research Network (2004). The high-level flow modeling used by SSF increases the performance of the simulator, but decreases its accuracy in comparison to packet-level simulators.

Yet Another Network Simulator is presented by Lacage and Henderson (2006). This simulator derives from a project to simulate MAC IEEE 802.11a/e in NS-2. Emulation is explored through the definition of a packet model that allows serialization, manipulation and fragmentation of packets in realistic fashion.

The simulation of distributed systems has different requirements and cannot be served well by low-level network simulators. For example, distributed applications with thousands of nodes are common in peer-to-peer systems. As network level simulators incur a great computational cost, they limit the growth of simulated scenarios to the scale found in real applications. This led to a new class of simulation systems (Naicken et al., 2007). According to Pujol-Ahulló et al. (2009), in this context, a simulator also needs to be modular, customizable and extensible, avoiding the restriction of scope of studied protocols due to implementation issues.

Neko (Urbán et al., 2002) is a distributed systems simulator whose main characteristic is code portability between simulation and the "real" environment. Authors claim that the set of methods employed in each case are the same, and that the choice is provided through an input file. Neko was developed in Java, achieving portability at JVM level (Lindholm and Yellin, 1999). The development of a protocol employing Neko's API is done in layers. According to this layered conception, the bottom layer will be connected to a component called *NekoProcess*. This component will abstract the communication between protocol layers under development and the *Network*. Each *NekoProcess* provides the option of a *passive* execution mode, whereby a *callback* method is employed to receive and process messages, or *active*, mode in which a thread receives messages and enqueues them. The *Network* is the communication medium employed by *NekoProcesses* for message transmission and it can be either simulated or real.

SimGrid (Legrand et al., 2003) is a simulator originally focused on the investigation of scheduling techniques for modern distributed platforms. Such studies require experiments with many environment variations, thus requiring a simulation tool with small execution times. This led the development of SimGrid to be focused in its execution speed. Fujiwara and Casanova (2007) argue that the simplifications adopted by SimGrid lead to inaccurate results in scenarios with small data sizes or networks with high contention. To tackle the problem, the authors propose the integration of SimGrid with GTNetS (Riley, 2003), a parallel version of NS-2 that focus on reduction of event list size, memory management, and reduction of the log file size. This integration

allows SimGrid to work also as a packet-level simulator, increasing its accuracy at the expense of performance.

PeerSim (Jelasity et al., 2009) is a simulator extensively used for the evaluation of peer-to-peer overlay networks. It presents a very modular architecture, allowing most of its components to be extended and fully configured through a simulation configuration file. As with other Java-based simulators, its engine is carefully optimized to offer small memory footprint and good performance, thus presenting good simulation scalability. PeerSim also presents features to allow the easy definition and observation of network topologies through graph abstractions, and the possibility to augmented simulation realism with the use of trace-based datasets for phenomena such as churn.

Another simulator focused on peer-to-peer systems is PlanetSim (Pujol-Ahulló et al., 2009). Its architecture follows a layered design with three levels. The upper layer implements the application that will communicate using the P2P overlay. The middle layer implements algorithms for initialization and maintenance of a few typical overlay networks. The lower layer, on its turn, allows the modeling of physical networks with different topologies, for evaluation of their impact in the simulated P2P applications. PlanetSim employs a set of behavior classes to control how overlay nodes respond to different kinds of events. *Behaviors* are special singleton classes activated when all attributes in a rule are met by an event processed by a node. This model allows node behavior to be modified without the necessity of modifications in its source code.

Simmcast differs from previous work in several aspects. Henderson et al. (2006), Nicol et al. (2003), Riley (2003), and Fujiwara and Casanova (2007) aim at abstraction and scalability meaning simulations with up to hundreds of thousands elements, at the expense of the simulation accuracy and detail. Simmcast, similarly to Pujol-Ahulló et al. (2009), focuses on extensibility, flexibility and the power to have a rich implementation that resembles the real implementation of the protocol. Simmcast is developed in Java, a mature language capable of supporting high performance applications (Taboada et al., 2009).

Simmcast also focuses on ease of simulation development. Its architecture presents a set of building blocks that closely resemble the components of a real network such as hosts, links and routers. To create a simulation, the developer extends these building blocks in order to implement the functionality of the protocol to be evaluated. Such an architecture allows the developer to create a simulation as if he/she were in fact implementing the protocol in a real network, easing the process of modeling the simulation. At the same time, the API of the building blocks hides from the developer issues related to simulation management, so he can focus exclusively in the details related to the protocol he wishes to simulate. The next section focuses on the description of the Simmcast architecture, presenting in detail each of its building blocks.

3. Architecture overview

The entities of the simulation and the relationship among them must be represented throughout the architecture in a consistent way. The proposed architecture does so by defining a framework where extensible *building blocks* are combined in order to describe the simulated network environment, and on top of it, the protocol under investigation.

The remainder of this section is organized as follows. An overview of the framework structure is presented (Section 3.1), followed by a description of the abstract building blocks and the interactions between them (Section 3.2). Routing of packets through a network topology is discussed in Section 3.3. Section 3.4 presents and explains the API, which includes primitives for sending and receiving messages. Finally, other aspects relevant to the architecture, such as the tracing interface and instantiation/execution are also discussed (Sections 3.5 and 3.6, respectively).

3.1. 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 (Schmidt and Buschmann, 2003). Besides, a framework is also advantageous because the architecture is not intended to a particular type of protocol or application, and thus derives 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 employed by other network simulation infrastructures, such as Jelasity et al. (2009) and Pujol-Ahulló et al. (2009).

The use of an actual framework in Simmcast design allows the simulator internals to be greatly simplified, since they are constructed defining what is called the *framework's core*. It comprises a simulation engine that provides processes and a kernel with basic packet-level communication and group management. The user, then, only needs to add or extend classes or interfaces of the framework to implement the desired protocol, simplifying the necessary effort to get the simulator ready to run.

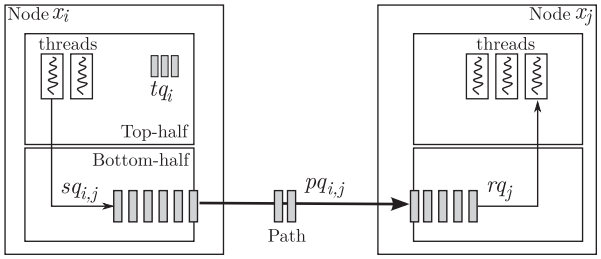
3.2. Building blocks

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 simulation, which led to the following building blocks: *node*, *thread*, *path*, *group*, *network*, *packet*, and *number 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. Their correspondence in the model is not dictated by the simulator: depending on the desired level of abstraction, nodes can represent a user agent, one end of a transport protocol in a host, a router. A node has two halves, *top* and *bottom*, containing user and Simmcast code, respectively.

In its top-half, a node will have one or more *threads* of execution that implement the logic of the protocol to be simulated (e.g., representing sender and receiver functionality). In general, 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.

As shown in Fig. 1, which depicts two neighbor nodes, the bottom-half of a node contains three *queues*: one to send out packets, another to receive in packets, and a third to record asynchronous events. An explanation about these queues follows. Let the set of existing nodes in a simulation be represented by x_1, x_2, ...,x_N, where N is the total number of nodes. A node x_i will have a sending queue $sq_{i,j}$ for any node x_j that x_i is connected to, 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_i). The capacity of sq and rq can be explicitly set or left unlimited. The $sq_{i,j}$ queue is served according to bandwidth to x_j, while rq_i is served according rate in which reception operations are invoked. Sending a packet means adding it to sq, an operation which takes t_{send} time; this



t_{qi} timeout queue for events from node i
$s_{qi,j}$ queue for messages to be sent from node i to j
$p_{qi,j}$ queue for messages intransit from nodes i to j
r_{qj} queue for messages to be received by node j

Fig. 1. Flow of messages between two nodes.

value can be used to limit the sending rate by the protocol. Taking a packet from rq through 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, such as *timeouts*. Timers can also be used to implement periodic processing behavior.

The classes that correspond to nodes and threads are *hotspots* in the framework. Several methods offer points of extensibility, being either abstract or empty. Since they are key factors during the development of a simulation experiment, these building blocks have a *white-box* characteristic, allowing even the most basic primitives from Simmcast (see later) 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 a user agent. Paths are used to *connect* nodes, and could represent a packet queue, a physical link or a logical path between two end-nodes. A path from x_i to x_j is represented by a path queue, or $pq_{i,j}$, that holds packets in transit from x_i to x_j.

A path has three attributes: 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.

Now we take advantage of Fig. 1 to provide a simple example of message transmission. First, node x_i sends a message to x_j; a copy of the message is enqueued in $sq_{i,j}$, space allowing. At this point, the sending thread at x_i is blocked for t_{send} time, if $t_{send} > 0$. The message in $sq_{i,j}$ eventually comes to the head of the queue, then waits for $p/b_{i,j}$ time, with p representing the packet size and $b_{i,j}$ the bandwidth of the path from x_i to x_j. Afterwards, the message leaves $sq_{i,j}$ and enters $pq_{i,j}$. The message stays in $pq_{i,j}$ for $d_{i,j}$ time, where $d_{i,j}$ is the latency delay of path from x_i to x_j. Then, the packet is enqueued at rq_j, space allowing; the message eventually arrives at the head of rq_j, and the next time a thread at x_j calls a receive operation, it will depart from rq_j. The service time is dictated by protocol logic, but can be no less than t_{recv}.

All classes that implement paths are *black-box*. The pq queues return their status to the user only through the trace mechanism, for accounting purposes. Management and resolution of paths are done at the kernel of the framework; user 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 class that represents a node implements a *Factory* design pattern (Gamma et al., 1995) for this end, interfaced through a command in the simulation description file.

Groups are relevant for several distributed protocols. Simmcast allows a group to be described either statically, through the experiment description file, or dynamically, through join and leave style primitives performed 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 packet is sent to g, and during its "propagation" x_i joins g, this message will not be delivered to x_i.

Packets (also called messages) are the unit of communication between any two or more nodes. The Packet class contains the minimal attributes required by a packet in Simmcast. Packets are containers to arbitrary objects, allowing packet types for new protocols to be easily defined either by inheritance or composition.

Each parameter of a simulation model can be of fixed or random nature. In Simmcast, they are read as *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, hyper-exponential, 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. 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 the *network*, a set of nodes in the simulated system that are combined and connected to form a *topology*. No specific routing scheme is enforced, so that different kinds can be used interchangeably (see next subsection for a discussion on routing). In fact, by varying the meaning of a node and its queues, simulations can cover a wide range of abstractions. In one extreme, the simulation of a P2P overlay may be comprised of a set of application processes connected through paths (reflecting TCP connections). In the other, a protocol simulation can be detailed to represent interactions among layers of network boxes (such as hosts, routers, switches). We call these schemes *abstract* and *concrete*, respectively, and a comparison between them is provided in Table 1.

Simmcast makes it easy to switch from an abstract to a concrete network models. In the simulation source code, it is only necessary to switch the extension of the implemented nodes from the Node to the HostNode abstract class (details about these

Table 1
Abstract vs. concrete views of network.

Building block	Abstract network	Concrete network
Node	Processes, nodes	Hosts, routers
Path	End-to-end paths and network clouds	Communication links
Group	–	Multicast groups

classes will be presented later). Thereafter, it is only necessary to alter the simulator configuration to include the desired router topology for the network (more information in Section 4).

The abstract view of the network does *not* include routing, only direct connections among nodes. So, essentially, a packet can only be transmitted from x_i to x_j if a path connecting these two nodes exists (that is, $\exists pq_{i,j}$). 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 ($\forall x_j \in g, \exists pq_{i,j}$). A higher level of detail can be easily obtained by making some of the nodes act like routers, by means of a concrete view of the network. This is an important design decision upon the simulation model, following the *Localized Cost principle*, choosing not to impose unnecessary complexity/processing cost.

To each of the previous building blocks, there is one corresponding class in the framework. Table 2 summarizes the building blocks and shows the corresponding class names.

3.3. Routing

When the abstract network view is employed, messages can be transmitted only between direct neighbors. Therefore, there is no need for routing. In contrast, when the concrete view is used, the topology comprises hosts and routers which will be connected arbitrarily. Thus, a message originated at a node may be destined to another that is several hops away: such a message will cross a set of routers, defined through a routing algorithm, ending in the destination host. Below, we focus on concrete networks and discuss routing issues.

With respect to routing, a simulator has the following requirements, which Simmcast aims to satisfy through its concrete network view:

- routing independence, that is, the logic of protocols or systems under investigation are unaffected by routing, allowing the use of an incremental approach;
- clear separation between the layer under investigation and the abstraction of the underlying layers;
- availability of arbitrary topologies, including rings, stars, trees, buses, or any combinations of those;
- generation of traces in different levels of abstraction, allowing the visualization of interactions among protocol or system elements (for example, through abstract or concrete views of the network).

Table 2
List of building blocks and corresponding framework classes.

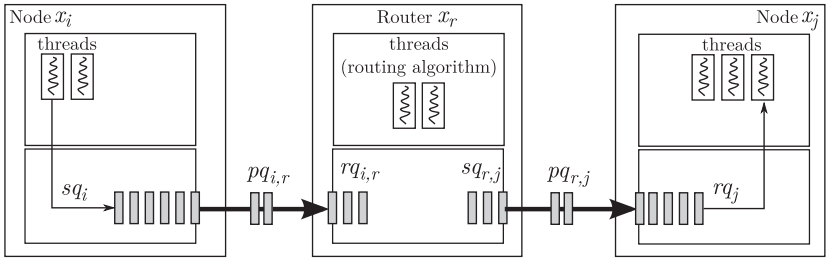
Building block	Class name
Node	Node
Thread	NodeThread
Path	Path
Group	Group
Number streams	RandomStream and subclasses
Packet or message	Packet

Table 3
List of building blocks and corresponding classes specific of concrete view.

Building block	Class name	Observation
Host node	HostNode	Transport or application protocol
Router node	RouterNode	Routing level protocol
Path	Path	Specialized, fixed propagation latency

Compared to the abstract view, the framework is changed in two ways. First, the *path* building block is overloaded with link behavior which, in practice, simplifies its behavior. Second, the concept of a node is extended to introduce two new building blocks: *host* node and *router* node, with corresponding subclasses. A router node implements a routing algorithm in its top-half, which will control message forwarding for its queues, maintained in the bottom-half. A host node, instead, receives transport and/or application code in its top-half. Table 3 summarizes the building blocks used in the creation of a concrete network model.

To illustrate these components, we augment Fig. 1 by adding the elements pertaining to the concrete network model. The result, presented in Fig. 2, shows the interactions during a message exchange between two nodes through a router. According to the properties previously pointed, the sending method of a host node x_i connected to a router x_r sends every packet through the sending queue sq_i, regardless of the final destination. The router extends the node by adding multiple receiving and sending queues, one for each incoming or outgoing link. In Fig. 2 example, the incoming link for x_i is associated with the queue $rq_{i,r}$, and the outgoing link for node x_j, with $sq_{r,j}$. In the context of host nodes, threads are used to represent the application/transport logic, while in router nodes, the routing protocol.



s_{qi} queue for messages to be sent by node i
$p_{qi,r}$ queue for messages in transit from node i to router r
$r_{qi,r}$ queue for messages to be received from node i by router r
$s_{qr,j}$ queue for messages to be sent from router r to node j
$p_{qr,j}$ queue for messages in transit from router r to node j
r_{qj} queue for messages to be received by node j

Fig. 2. Message flow between two nodes in a concrete network model.

Packets are forwarded by routers according to a routing table. This table may be filled statically (in the beginning of the simulation) or dynamically (through a routing protocol). These two approaches will be discussed next.

Static routing may be employed in simulations where the interactions among routers can be abstracted and routing information will not change during simulation. In this model, a universal routing table (including multicast information) is established a priori. For multicast, the default scheme is SPT (*Shortest Path Tree*), but this can be extended/changed as desired. The convergence time is zero, since the table is static and any changes to the topology are immediately reflected on it.

Although static routing is expected to be sufficient for most environments, it is possible that a more detailed simulation of the network layer will be desired. This may be so because the transport/application logic will be substantially affected by routing or because the interest is on topology changes and how they will reflect on the protocol under investigation. In such cases, dynamic routing should be used, since it allows a more realistic view of the network. To that end, a router node can be extended into a dynamic router node, containing one or more threads, which execute the routing algorithm. Each dynamic router node contains its own routing table, and is responsible for maintaining it consistently. This is achieved by means of control messages exchanged among routers, according to the adopted protocol.

3.4. API

The simulation interface is defined as an API with typical communication and timer operations, as well as concurrent software architecture suitable for designing simulated group communication protocols and applications. The primitives of the simulator are no more intrusive in the designer's code than actual system calls would be. It is in fact often less intrusive, since simulated code usually deals with a lesser number of exception cases, and does not have to deal with synchronization among threads in a node, since the discrete-time model implements cooperative multithreading.

Simmcast offers a series of methods that define a concise set of primitives, as listed in Table 4, 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.

Perhaps the most important primitives for a protocol are send() and receive(). The former is non-blocking: the message is enqueued for transmission, space allowing, but the calling thread gets no feedback about the success of this operation. The receive() primitive, on its turn, requests the receipt of a message and is blocking: if no message is available in the receive queue, the calling thread is blocked until one arrives or a specified timeout expires. Primitives join() and leave() regard groups, and are implemented in the Group class, whose instances represent the active groups in the network. The primitives setTimer(), onTimer() and cancelTimer() are used respectively to set a timer, to handle an expired timer, and to cancel a pending timer. These allow the developer to trigger as many events as necessary by its implementation, handling them by customizing the onTimer() method implementation. Finally, the primitives sleep() and wakeUp() are related to scheduling, being used respectively to put the calling thread to sleep (either indefinitely or for a specified time) and to awaken another thread which is sleeping.

Table 4
List of primitives.

Name	Action
send()	Sends a packet/message to a given destination (does not block)
receive()	Blocks until a packet is available (a timeout may be provided)
tryReceive()	Attempts to receive a packet or returns "null" (non-blocking)
join()	Node joins a given multicast group
leave()	Node leaves a given multicast group
setTimer()	Configures timer to expire in a given time
cancelTimer()	Cancels an existing timer
onTimer()	Method to be invoked when a timer expires
sleep()	Put the current thread to sleep
wakeUp()	Wake up another thread that may be sleeping

3.5. Simulation output and traces

Several different metrics are used in protocol simulation, depending on three factors: the nature of the experiment, the level of abstraction, and the kind of protocol evaluated. Some examples of metrics 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. However, when application-level protocols are being investigated, metrics are less typical. Hence, flexibility is very important. Translated into the Simmcast architecture, 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 a queue (*sq*, *pq*, *rq*, and *tq*). For example, according to Fig. 1, 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 future asynchronous events 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 an unified output interface to where all events in the simulation are reported through a special class. This class, named TraceGenerator, has a read-only view of the entire simulation scenario, and reports information that 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. The tracing class 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 tracing subclass that generates native traces using this interface. The user may also subclass TraceGenerator in order to perform any desired custom accounting or output data in other formats.

3.6. Instantiation and execution

Unlike the conventional approach of using a simulator (or building one from scratch), a simulation experiment with Simmcast is done in two stages, as follows. First, it is necessary to instantiate the framework, constructing a new protocol by combining existing building blocks and specializing the classes that represent nodes. Depending on the protocol investigated, there may be one or more kinds of agents, such as peer agents, replica managers, sender, receiver, client, server, master, slave, and so on.

The second stage is to execute the resulting framework instantiation in a given scenario. A scenario will be specified through a set of different types of nodes and their connections, thus forming the network topology. 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. 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. 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 to specifying the topology and startup parameters.

4. Implementation aspects

This section briefly discusses design and implementation aspects from the instantiation of the architecture previously overviewed. The latest implementation of Simmcast follows a layered design with five levels, allowing a better separation of the framework components, which also facilitates its manipulation for creating simulations:

Layer 1: Implements the Simmcast engine, which is a generic discrete-event machine based solely on the concept of processes. This layer is not visible to the user.

Layer 2: This layer implements the kernel of the framework, composed by fundamental classes described in Section 3.2 such as Node, NodeThread, and Path.

Layer 3: This layer implements the routing mechanisms necessary for concrete network models, such as the HostNode and RouterNode, and also necessary routing algorithms. These entities are implemented through extension of layer 2 elements, and also can be extended according to the necessity of the user.

Layer 4: This layer encompasses the implementation of the protocol that will be simulated through the framework. It is composed by extensions of elements of layers 2 and 3 such as nodes and threads, augmented to implement the intended protocol logic.

Layer 5: The final layer encompasses the parametrization of simulation entities such as configuration of nodes and network topology. It is specified through a text file loaded in the simulation initialization.

The Engine implements the discrete-event machine responsible for thread scheduling in the framework. Because experiments need to be reproducible, the scheduling is *cooperative* instead of *competitive*. A single thread runs each time and is never preempted. Internally, the Engine implements this model through monitors, a synchronizing mechanism available in Java. In these monitors, threads synchronize by cooperation through wait() and notify() of Object class, with one object per thread. A semaphore is employed with each monitor, to guarantee that the corresponding object is being blocked/unblocked once (Venners, 1999).

Processes, which are implemented through a thread, are the main component of the engine. Their life-cycle follows the model illustrated by Fig. 3. As shown, a process may be in one of five different states: ready (to run), running, sleeping (blocked), joining (waiting for another process), and ended (terminated). Each state transition is associated with a method implemented in the core process class.

The main structure associated with process scheduling by the engine is the process queue. It contains all processes that are scheduled to run in some time t, equals the current time or larger. Abstractly, it is a priority queue with tuples which store time and process, (t,p), ordered by t, the scheduling time. Processes with the same t are ordered according to their "arrival". For example, if a running process p causes a process q_1 to be scheduled to run at time t, and immediately after that a process q_2 also to run at time t, the priority queue will have first q_1 and then q_2. Each simulation event will have one given process designated to handle it. Hence, the process queue corresponds to the "time wheel" of the simulator.

Internally, the process queue is implemented through a TreeMap class, which is implemented (by the Java Virtual Machine) as a Red-Black Tree as described by Cormen et al. (2001). There are two operations defined for this queue. p.insertAt(t) inserts p in the priority queue, with scheduling time t. removeFirst() takes the head of the queue, say r, which is the next process to be executed by the Engine, and advances the simulation clock to the time t associated with r if it is larger than the current time. The algorithms of both methods are expected to have a complexity of $O(\log n)$, where n is the number of processes currently scheduled for execution.

5. Performance evaluation

The implementation described in the previous section was benchmarked to assess the individual costs in fundamental Simmcast operations. Based on these costs, we were able to establish lower bounds for processing time and memory consumption. This section discusses the results of the benchmark and other performance issues.

The performance of a simulation in Simmcast will be broadly affected by the following aspects: (a) the complexity of the target distributed algorithm or protocol under simulation; (b) system size, that is, the number of nodes and threads in the simulation; and (c) the overhead added by Simmcast.

The first aspect is independent of Simmcast and of any simulator: if the distributed protocol under study is computationally- or message-intensive, then the results will take longer to come out in the simulated one as they would in real life. Besides, the richer in detail is the simulation model, the more expensive it

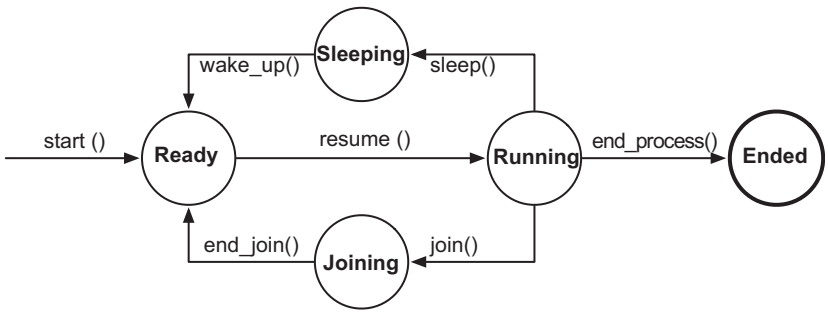


Fig. 3. Process state diagram.

will be to run (but more accurate will be the results). The ability to represent in Simmcast the protocol logic with different levels of abstraction and detail helps mitigating this cost.

With respect to the second aspect, if all nodes are hosted within the same machine hardware, then there is a natural slowdown: the protocol logic of each node and every message exchanged will be carried out within the same hardware. This is inherent to sequential simulation and a limitation of Simmcast. It was a design decision, though, to avoid complexity and keep the framework small and simple.

The third aspect, overhead, is related to the simulator efficiency and, hence, the focus of the analysis presented in this section. The results shown below are based on experiments executed in a computer with a Intel Core2 CPU running at 2.4 GHz and 8 GB of RAM.

5.1. Basic engine operations

To determine the most relevant operations executed by Simmcast kernel during a simulation, we observed its methods invocation frequency. The results showed, as expected, that the dominant function performed by the kernel is the scheduling and execution of simulation events created by processes. Such a function is conducted through the methods insertAt and removeFirst, implemented in the TimeWheel class.

Recall from Section 4 that the time wheel class maintains a red-black tree containing all simulation events scheduled for execution, indexed by their activation time. The method insertAt is used to schedule a new event for execution. The costliest operation in this method is the insertion of the event in the red-black tree, which has complexity $O(\log n)$. The method removeFirst, in turn, is used to get the next event to be executed by the simulator. The operation of highest complexity in this method is the removal of the process indexed by the smallest key from the tree, which is also $O(\log n)$.

To further analyze the performance of the event scheduler, the execution time of these two operations was measured while scheduling and executing 1 million events. The results are shown as scatter plots in Fig. 4, where the vertical axis represents the execution time in microseconds (μs), and the horizontal axis, the number of events already contained in the time wheel. Each plotted dot represents the average time for 1000 executions of the operation.

In both cases, the execution times follow the expected behavior of $O(\log n)$ complexity. This can be clearly observed for the insertAt in Fig. 4(a), where the execution time rapidly grows 1.2 μs for the first 200,000 events, while the growth is of only 0.2 μs for the last 200,000. In all cases, the average execution time of insertAt always stays bellow the 4 μs. The method removeFirst execution time, presented in Fig. 4(b), stays below 1.5 μs. It presents a nearly constant execution time in the majority of observed cases, but the $O(\log n)$ behavior can be observed in the removal of the last 100,000 events, where the variation of the execution time is of 0.2 μs.

5.2. Simulation performance

There are two simulation phases to consider: setup and execution. The setup of a simulation will start with the parsing of the configuration file, when the structures associated with the topology will be created. That is, nodes and paths will be instantiated, as well as associated threads. Once the topology has been created, the configuration file will typically contain calls made to public methods of topology objects, mainly nodes. All instantiation and parametrization of the simulated system can take place during the configuration file parsing.

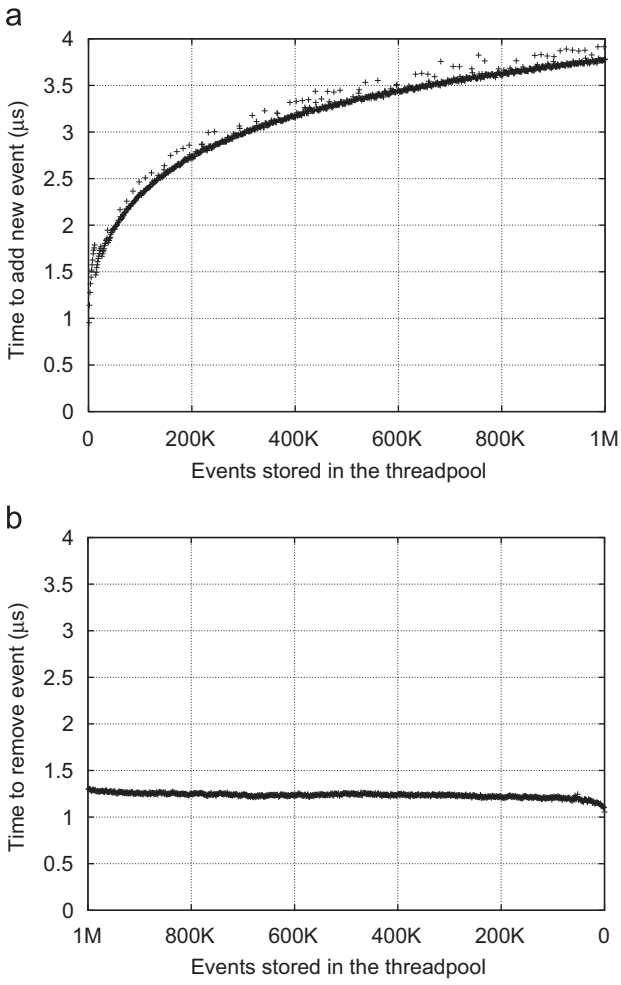


Fig. 4. TimeWheel methods performance measurement: (a) event insertion (insertAt) and (b) event removal (removeFirst).

In this phase, two operations are dominant: creation of nodes, and creation of paths among them. To assess the general performance of the simulation in its configuration step, we measured the execution time to set up a simulation with 100,000 nodes and 100,000 paths between randomly chosen pairs of nodes. The results are shown in Fig. 5, where the vertical axis represents the measured time in microseconds (μs), and the horizontal axis corresponds to the number of elements already created. Each dot represents the average time for 100 executions of the instruction.

The results in Fig. 5 show that, on average, the execution time remains constant regardless of the number of existing nodes and paths. Figures 5(a) and (b) indicate, respectively, that the creation of a node takes about 9 μs, and of a path, about 2 μs. Nodes are managed in the simulation through various data structures, which are manipulated when new nodes are created. Such processing leads to the higher overhead observed in the creation of the first 10,000 nodes. Data structures are also manipulated in the creation of new paths, but since they are much simpler, no considerable overhead can be observed.

Another metric of interest is memory consumption. In the setup phase of a simulation, some classes are heavily instantiated, with the creation of large numbers of objects. We analyzed a dump of the Java VM heap obtained after the setup phase of a simulation (that is, after the processing of the configuration file), and identified nodes, threads and paths as the dominant classes. Table 5 summarizes the memory overhead for each of these classes, per element.

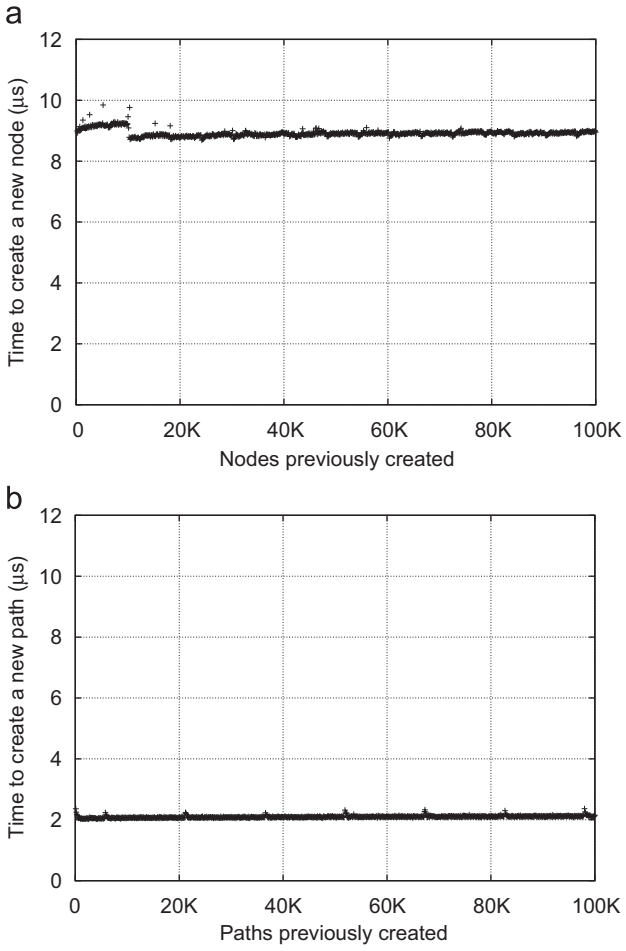


Fig. 5. Simulation setup performance measurement: (a) node creation and (b) path creation.

Table 5
Memory overhead of the dominant simulation classes, per instance.

Class	Memory (Bytes)
Node	136
NodeThread	240
Path	104

The values presented in Table 5 for the Node and NodeThread classes refer to bare classes, that is, there are no other structures or logic implemented within them. When developing a simulation with Simmcast, these classes will be extended to reflect the logic of the protocol to be simulated, and thus consume more memory. The topology of the simulated network is another factor to directly influence the amount of memory, because there will be different numbers of paths that must be instantiated for different topologies.

In the execution phase of the simulation, the dominant operations are message transmission and reception. In the former case, a message will be dispatched by the sender node and added to the receiving queue of the destination node. In the latter, a message can be removed from the receiving queue and returned to the receiver node. A family of sending and receiving methods are implemented in the Node abstract class.

To assess the performance of the methods and structures involved in message processing, three test scenarios were evaluated. The first one involves two nodes, a source and a sink, which will exchange a fixed number of messages. Source and sink are connected through one router in a *star topology*, so that messages will always cross only two hops. A variable number of dummy nodes were connected to the router, so that we could measure their influence in the message routing times.

The second scenario also consists of a source and a sink, but now they are separated by a *chain* of interconnected routers. The number of routers was varied with each new run. With larger number of routers, we expect to see an increase in the simulation times, due to the greater number of hops which the messages need to traverse, and due to the overhead caused by the methods that control routing tables at routers.

The third scenario consists of a *mesh* topology with a variable number of nodes. In this scenario, two random nodes exchange a fixed number of messages, while the others remain inactive. We expect this scenario to show nearly constant results, since the messages will always travel through one hop in the network. Another difference in this scenario is that it is implemented with an abstract network model, unlike the previous ones (which are concrete).

We executed the above-mentioned scenarios with topologies between 10 and 1000 elements. In each execution, the source node created 1000 messages and sent them to the sink node, which just received and discarded the messages. Figure 6 shows, in all scenarios, the average time cost for transmission of one message. The horizontal axis represents the number of elements in the topology, while the vertical axis represents the simulation time in microseconds.

Figure 6 indicates that both star and mesh topologies have a near constant time cost of 17 µs for the mesh topology and 28 µs for the star topology, regardless of the number of nodes present in the network. The mesh topology time cost is smaller because it is implemented as an abstract network model, avoiding the overhead related to routing algorithms present in the concrete network model. Moreover, messages are sent through only one path in the mesh topology, while they must be sent through two paths in the star topology, increasing the overhead in the latter. The time cost for the chain topology presents a constant behavior for small topologies, with less than 20 nodes, with a time cost of 80 µs, presenting as expected a linear growth for larger topologies, with an added overhead of 1 µs for each router added to the network.

6. Framework instantiation

This section presents case studies in which Simmcast was employed for the investigation of various application-layer protocols. Such studies include framework instantiations to evaluate

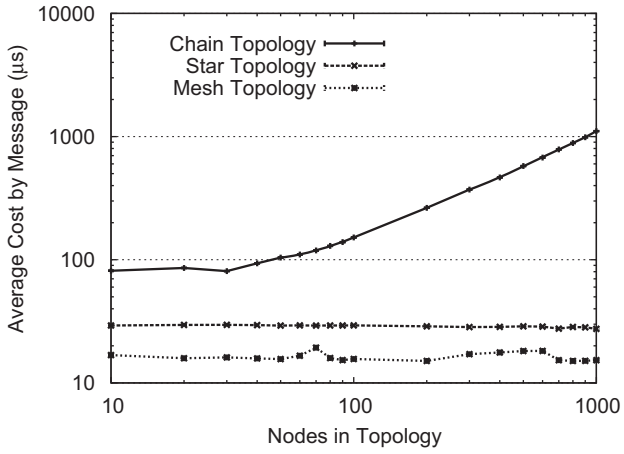


Fig. 6. Simulation performance for three common topologies.

denial of service attacks against BitTorrent (Section 6.1), pollution avoidance mechanisms in file sharing (Section 6.2), and a protocol for secure service discovery in ubiquitous computing environments (Section 6.3). To further demonstrate its extensibility, we present an integrated API that enables the execution of both simulation and "live" experiments using the same piece of protocol source code (Section 6.4).

These case studies provide only an overview of how Simmcast can be used to evaluate different kinds of application-layer protocols. The interested reader will find, in the following subsections, references that describe in greater detail each of the presented instantiations.

6.1. Investigating DoS attacks against BitTorrent networks

BitTorrent, Inc. (2010) is the most popular P2P file sharing protocol in use today, with millions of users (Ipoque, 2010). Its security, as well as with other peer-to-peer protocols, is an open research challenge. The study by Konrath et al. (2007) presents an evaluation of the impact of attacks that exploit BitTorrent vulnerabilities with the sole intention of harming a swarm. It is further extended by Barcellos et al. (2008a), with the proposal of two countermeasures to effectively tackle three attacks. Both studies were conducted using TorrentSim, an instantiation of Simmcast to implement the logic of BitTorrent user agent and tracker by extending the framework classes.

In BitTorrent, content is described by a torrent file, which includes information about one or more *trackers*. A swarm is composed by a group of peers interested in sharing some content and which connect to each other forming a network with random topology. A tracker is responsible for keeping an updated list of peers participating in the swarm. TorrentSim adopts a concrete network model to represent a swarm with its peers and trackers; both are created by extending the HostNode class. These can be further specialized into honest or malicious peers. Each peer contains multiple threads, one for each remote peer connected. The overlay topology is arbitrarily formed during the simulation, according to the connections established among peers. Using a full mesh as underlying topology in this case would be very inefficient, since very few connections would be in fact ever used. Therefore, a star topology is used, with a RouterNode at the center and HostNode instances neighboring it. Paths between HostNode instances and the RouterNode are set with random delays and no losses to reflect the abstraction provided by a TCP connection. A comparison of simulation results and traces from a controlled BitTorrent environment showed that the simulated swarm behavior is very similar to that of real networks.

The success with TorrentSim led to the development of TorrentLab, a testbed to evaluate BitTorrent networks through simulations and live experiments (Barcellos et al., 2008b). TorrentLab provides a modeling environment which allows a scenario with a swarm to be easily described. This scenario can be used as input to a simulation and/or to a live experiment, to be performed, respectively, by modules called TorrentSim and TorrentExp. The data they generate is processed through a result consolidation environment responsible for log analysis, consolidation and plotting. TorrentExp executes live experiments through a weakly synchronized instantiation of BitTorrent user agents and a tracker over a computer network. The instantiation and control of these entities is entirely managed by TorrentExp. TorrentLab can also take advantage of parallel execution through grid environments to accelerate the completion of TorrentSim simulated scenarios.

6.2. Studying content pollution

Content pollution is one of the major threats to file sharing P2P applications. Various research efforts aim to understand the dynamics of this kind of threat and to propose means to identify polluted content. The study by Barcellos et al. (2011) presents a pollution control strategy with focus in the early stages of content dissemination. The number of allowed downloads of a content is adjusted according to a reputation metric, which is determined according to user feedback.

Four pollution control mechanisms were proposed based on the strategy, following classic distributed systems designs. Three of these mechanisms depend on a download manager, which receives feedback from peers that finished downloading a content, calculates reputation, and issues download grants. The difference among the three approaches lies in the number and organization of managers in the network topology. The fourth mechanism is totally decentralized, and peers themselves decide if they can begin downloading based on queries to other peers.

The four mechanisms were evaluated through an extension of the framework. Nodes are extended to represent peers, either correct or malicious ones, and the download manager. Each mechanism was implemented by extending the base classes of the strategy to incorporate the functionality specific to each design. In the conducted study, we focused on the performance evaluation of the mechanisms regardless of network topology. The three strategies that depend on the download manager use a concrete network model with a star topology, due to its simplicity. The topology was composed by a RouterNode at the center, interconnecting instances of peers and download managers. The decentralized strategy, on its turn, assumes a topology with no specific organization, where each peer connects with a random number of neighbors. As peers themselves are responsible to propagate queries to their connections, the Simmcast routing layer (as presented in Section 4) was not necessary. Thus, this strategy uses an abstract network model, where each peer is randomly connected to a small number of neighbors (a parameter in the configuration file). Variables dependent on different probabilistic distributions, like peer arrival and content download times, were implemented straightforwardly through the RandomStream classes provided by Simmcast.

6.3. Secure service discovery in ubiquitous computing

A service discovery protocol is an important element of an ubiquitous computing environment. Such service should help peers finding the available services in an environment, and at the same time avoid the exposition of sensitive information that could be used to launch attacks against clients. Flexible Secure Service Discovery (FSSD), presented by Moschetta et al. (2010), is a protocol that allow users to define the levels of security and privacy when collaborating with the service discovery mechanism. Peers employing the protocol use a trust network, built according previous interactions among them. Messages are only propagated to peers with a trust relationship higher than the confidence levels specified in each message.

A Simmcast extension was developed to evaluate the protocol. Each node implements the functionality of a device in a ubiquitous environment, acting both as a provider, periodically announcing a service in the network, and as a client, launching queries for other services. The FSSD protocol is implemented as a library, independently from the Simmcast API, to allow future deployment in real devices. To simulate FSSD, the library was integrated to the protocol layer of the developed extension (see Section 4). The configuration layer is used to set up the trust network among nodes, through an algorithm that creates a topology with small world properties, commonly found in such networks. The generated topology is written to the configuration file loaded in the Simmcast initialization. It should be noted that the trust network is a component maintained internally by the FSSD protocol library,

and thus its connections are independent of the underlying network configuration. One of the goals of our evaluation was to study the influence of the trust network topology on protocol performance, without interference from the underlying communication network. Thus, to keep it small and simple, a concrete network model with a star topology used, with end-to-end connection properties (e.g., ordered delivery with random delays) being simulated through the behavior of a configured Path.

6.4. Uniting simulation and live experiments

To achieve better insights in the analysis of complex network systems, more than one evaluation method should be used. Simulation and live experiments are two alternatives that can be combined in many cases, but often do not appear together due the implementation effort associated with each method. The differences between simulation and prototype code usually lead to these methods being used in different steps of the analysis. Instead, simulation and experimentation could be part of the same, integrated set, such that the researcher could seamlessly alternate between them.

Simmcast Testbed (Barcellos et al., 2006) allows extensions of Simmcast to be used also for the execution of live experiments in a real network. It is an API for the Java language, comprehending communication primitives and a thread model, allowing the development of code to be used both for simulation and live experimentation. The API abstracts peculiarities found on each environment, and possesses two implementations: Testbed-Sim, which uses as back-end the Simmcast framework; and Testbed-Exp, which uses the Java thread and network APIs.

The Simmcast Testbed thread model follows the one adopted by Java, and comprised two classes: ProgramThread, which corresponds to a Java thread, having an interface very similar to the later; and Program, corresponding to an independent instance of a program, including the main method through which other threads are initiated. The Testbed also implements communication primitives that are in an intermediary level of abstraction between Simmcast and Java programming models. These primitives include methods to send and receive messages, both through unicast and multicast channels.

Testbed-Exp internally manages low-level network operations, like socket openings and multicast group configuration. Network addresses are also managed internally, abstracted to simple integer identifiers through the NetworkAddress class. These abstractions allow the Testbed API to keep a strong correspondency with both the Java API programming model, as well as with the model employed with Simmcast.

The execution model choice (simulated or real) happens at runtime, by specifying in the experiment configuration which package from simmcast.testbed should be loaded. This allows the alternation between Testbed-Sim and Testbed-Exp without recompilation. Apart from initialization, which requires small adaptations to the main routine of the experiment, all protocol code used in Testbed-Exp and Testbed-Sim is identical.

7. Design lessons and concluding remarks

This paper presented Simmcast, an extensible object-oriented simulation framework for application-layer protocols. The flexibility of Simmcast allows a simulation to be gradually developed until it achieves close resemblance to a real protocol implementation. It also lets the researcher's focus to be kept in questions related to the studied protocol: while it abstracts the issues of low-level network layers, present in more complex simulators, it also provides the essential programming abstraction of nodes, threads and messages.

Simmcast takes advantage of the reusability and generality offered by framework concepts to ease the implementation of diverse classes of protocols. The API contains building blocks that closely resemble components of a real network, such as user agents, hosts, routers and messages. Simulations can be created by simply extending these components so they act according to the protocol to be evaluated. Such an architecture allows the developer to create a simulation as if he were in fact implementing the protocol in a real network, facilitating the process of modeling the simulation. The abstraction provided by the building blocks also hides lower level simulation control issues from the protocol implementation. This allows the developer to keep his attention in the more important task of modeling and evaluating the application-layer protocol.

The framework architecture follows a layered design. Lower levels, which contain the Java virtual machine and process execution engine, are hidden from the user implemented protocol, located at the upper layers. Middle layers comprehend the framework's building blocks, the user's implemented protocol, and the routing topology for communication among nodes. The upper layer comprehends the configuration of environment parameters for one experiment, described through a text file of simple syntax, allowing the evaluation of different parameter configurations without the necessity of modifications in the implementation of simulated entities.

Simmcast allows the creation of a model with incremental level of detail. First versions of a simulation can be developed through an abstract network model, where nodes represent applications connected through an end-to-end communication channel. This model can be gradually extended to a concrete network model, which will allow the evaluation to be conducted with routing in different topology configurations.

The extensibility offered by the framework was a key factor in the development of simulations for the evaluation of various kinds of application-layer protocols, some of them described in Section 6. The layered architecture allows the development to focus on the issues related to the protocol to be implemented, leading to simulations with clean and extensible design. Further analysis (presented by Konrath et al., 2007) demonstrated that a simulation developed through Simmcast present a behavior very close to that found on an experiment in a controlled environment. This leads to simulations that offer a great deal of insight about evaluated protocols, without necessity to deploy complex experimental environments.

Acknowledgments

This work has been partly supported by Brazilian research funding agency CNPq – Conselho Nacional de Desenvolvimento CientÃ fico e Tecnológico, under the CT-Info Project Grant no. 552178/2002-0. Part of this research was conducted while the first author was with Unisinos University, Brazil (2001–2007).

References

Bajaj S, Breslau L, Estrin D, Fall K, Floyd S, Haldar P, et al. Improving Simulation for Network Research. Technical Report. USC Computer Science Department; 1999.

Barcellos MP, Bauermann D, Sant'anna H, Lehmann M, Mansilha R. Protecting bittorrent: design and evaluation of effective countermeasures against dos attacks. In: SRDS'08, 27th International Symposium on Reliable Distributed Systems, Napoli; 2008. p. 73–82.

Barcellos MP, Facchini G, Muhammad HH, Bedin GB, Luft P. Bridging the gap between simulation and experimental evaluation in computer networks. In: 39th Annual Simulation Symposium, Huntsville; 2006. p. 1–8.

Barcellos MP, Gaspary LP, Cordeiro WLdC, Antunes RS. A conservative strategy to protect P2P file sharing systems from pollution attacks. Concurrency and Computation Practice & Experience 2011;23:117–41.

Barcellos MP, Mansilha RB, Brasileiro FV. Torrentlab: investigating bittorrent through simulation and live experiments. In: ISCC'08, IEEE Symposium on Computers and Communications, Marrakech; 2008. p. 507–512.

BitTorrent, Inc., BitTorrent. 〈http://www.bittorrent.com〉; 2010.

Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 2nd ed. MIT Press; 2001.

Cowie J, Nicol DM, Ogielski AT. Modeling the global internet. Computing in Science & Engineering 1999;1:42–50.

Fujiwara K, Casanova H. Speed and accuracy of network simulation in the SimGrid framework. In: ValueTools'07, 2nd international conference on performance evaluation methodologies and tools, Nantes; 2007. p. 1–10.

Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional; 1995.

Henderson TR, Roy S, Floyd S, Riley GF. NS-3 project goals. In: WNS2'06, 2006 workshop on NS-2: the IP network simulator, Pisa; 2006.

Ipoque. Internet study 2008/2009. 〈http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009〉; 2010.

Jelasity M, Montresor A, Jesi GP, Voulgaris S. Peersim: A peer-to-peer simulator. 〈http://peersim.sourceforge.net/〉; 2009.

Konrath MA, Barcellos MP, Mansilha RB. Attacking a swarm with a band of liars: evaluating the impact of attacks on bittorrent. In: P2P'07, 7th IEEE international conference on peer-to-peer computing, Galway; 2007. p. 37–44.

Lacage M, Henderson TR. Yet another network simulator. In: WNS2'06, 2006 workshop on NS-2: the IP network simulator, Pisa; 2006. p. 650–7.

Legrand A, Marchal L, Casanova H. Scheduling distributed applications: the SimGrid simulation framework. In: CCGrid'03, 3rd IEEE/ACM international symposium on cluster computing and the grid, Tokyo; 2003. p. 138–45.

Lindholm T, Yellin F. The Java(TM) virtual machine specification. 2nd ed. Prentice Hall PTR; 1999.

Moschetta E, Antunes RS, Barcellos MP. Flexible and secure service discovery in ubiquitous computing. Journal of Network and Computer Applications 2010;33:128–40.

Muhammad HH, Barcellos MP. Simulation group communication protocols through an object-oriented framework. In: ANSS'02, 35th annual simulation symposium, San Diego; 2002. p. 143–50.

Naicken S, Livingston B, Basu A, Rodhetbhai S, Wakeman I, Chalmers D. The state of peer-to-peer simulators and simulations. ACM SIGCOMM Computer Communication Review 2007;37:95–8.

Nicol DM, Liu J, Liljenstam M, Yan G. Simulation of large scale networks 1: simulation of large-scale networks using SSF. In: WSC'03, 35th winter simulation conference, New Orleans; 2003. p. 650–7.

NS-2 Developers. The network simulator NS-2. 〈http://www.isi.edu/nsnam/ns/〉; 2009.

NS-3 Developers. The NS-3 network simulator. 〈http://www.nsnam.org〉.

Nygaard K, Dahl OJ. The development of the SIMULA languages. In: History of programming languages, vol. 1. New York: ACM; 1978. p. 39–480.

Pujol-Ahulló J, García-López P, Sànchez-Artigas M, Arrufat-Arias M. An extensible simulation tool for overlay networks and services. In: SAC'09, 2009 ACM symposium on applied computing, Honolulu; 2009. p. 2072–6.

Riley GF. Simulation of large scale networks 2: large-scale network simulations with GTNetS. In: WSC'03, 35th winter simulation conference, New Orleans; 2003. p. 676–4.

Schmidt DC, Buschmann F. Patterns, frameworks, and middleware: their synergistic relationships. In: ICSE'03, 25th international conference on software engineering, Portland; 2003. p. 694–704.

SSF Research Network. SSF website. 〈http://www.ssfnet.org〉; 2004.

Taboada GL, Tourino J, Doallo R. Java for high performance computing: assessment of current research and practice. In: PPPJ'09, 7th international conference on principles and practice of programming in Java, Alberta; 2009. p. 30–9.

Urbán P, Défago X, Schiper A. Neko: a single environment to simulate and prototype distributed algorithms. Journal of Information Science and Engineering 2002;18:981–97.

Venners B. Inside the Java 2 virtual machine. 2nd ed. McGraw-Hill; 1999.