An extensible simulation tool for overlay networks and services[*]

Jordi Pujol-Ahulló, Pedro García-López, Marc Sànchez-Artigas, Marcel Arrufat-Arias
{jordi.pujol, pedro.garcia, marc.sanchez, marcel.arrufat}@urv.cat
Universitat Rovira i Virgili, Av. Països Catalans, 26. 43007 - Tarragona, Spain

ABSTRACT

Research community on distributed systems, and in particular on peer-to-peer systems, needs tools for evaluating their own protocols and services, as well as against other protocols with the same preconditions. Since a (TCP/IP) experimental evaluation is not always feasible, simulation tools appeared.

In this paper we introduce PlanetSim, a discrete event-based simulation framework tool for overlay networks and services, as well as extensions from third parties that prove its true extensibility and adaptability to the researchers' needs. In addition, we introduce within PlanetSim a novel methodology for implementing peer-to-peer overlay protocols based on *behaviors*.

Categories and Subject Descriptors

I.6.8 [**Types of Simulation**]: Discrete event
; D.4.8 [**Performance**]: Simulation, Measurements; D.3.3 [**Language Constructs and Features**]: Frameworks

General Terms

Algorithms, Measurement, Performance, Design

Keywords

Discrete event-based simulation, overlay network simulation, frameworks, design patterns.

1. MOTIVATION

From the appearance of distributed systems, and in particular peer-to-peer systems, the research community needs tools for evaluating their own protocols and services and, even more important, comparing their works against other protocols *with the same preconditions*. Two possibilities were proposed: *experimental evaluation* where prototypes of those protocols are tested in real testbeds, like PlanetLab [5], and *simulated evaluation* where protocols are analyzed with some network settings assumptions. As a consequence of node instability within those testbeds, lots of simulation tools (*simulators*) have been appearing to help the research community, becoming standard platforms where different works are analyzed. Nevertheless, the vast majority of them are ad-hoc customized simulators [11] and they are not for general overlay evaluation purposes, poorly documented or not extensible to other protocols and settings. Thus, in this paper we are mainly interested in extensible, scalable, high-level overlay and services simulators. In particular, we focus on simulators that supports structured (e.g., Chord [15], Pastry [14]) and unstructured (e.g., Gnutella [1]) peer-to-peer systems and services simulation.

This way, we believe that the following points are the challenges to deal with in order to develop an appropriate overlay protocol and services simulation tool.

High-level simulation for large-scale network evaluations. It is obvious that the resources in computers are limited. Thus, there exists a *tradeoff on the simulation precision*. Clearly, packet-level simulators (e.g., NS-2 [7], OMNET++ [2]) have a true high cost in time and computer resources, thus not scaling to thousands of nodes. Nevertheless, high-level simulators (e.g., p2psim [3], PeerSim [4], FreePastry [14], PlanetSim [10]) show a better performance, and thus, enabling big scale network evaluations. In addition, notice peer-to-peer researchers are usually more interested in algorithm verification than in simulating the full TCP/IP stack.

Convenient overlay network extensibility. We believe that one of the most important features for an overlay simulator is its capability to enable easily and gracefully the development of new overlay protocols, as well as the possibility to run services (like Scribe [8]) on top of them, regardless the overlay protocol (i.e., via the Common API [9]). It is even more important because the simulator can provide lots of functions and researchers do not need to know them at the whole. To this end, p2psim does not support application level extensions and, in the other way around, FreePastry is not extensible to other overlay protocols.

Modular, customizable and extensible. While the overlay research field is huge, such a simulator cannot be restrictive on its provided functionality. Instead, we believe

[*]This work has been partially funded through project P2PGRID, TIN2007-68050-C03-03, and FPU National Program, AP-2006-04166, of the Spanish Ministry of Education and Science.

that a simulator designed to follow well-known software engineering techniques will clearly help researchers to learn and extend the simulator as necessary. A good documentation is another key point to guarantee its extensibility. For instance, PeerSim provides a lot of functionality, but this simulator is designed in a static, ad-hoc way, becoming difficult to extend it in an easy and clear way.

Gathering and showing meaningful information. Retrieving simulation results for their later analysis is a central point within this kind of simulators. It is clear that there exists a *tradeoff between the precision of results gathered and an expected time-efficient simulation.* While some researchers will be interested in gathering some basic statistic information (like the total simulation time), some others will be interested in gathering a low-level statistic information (like protocol message traffic between nodes). We believe that such a simulator should be flexible and should provide both kinds of simulation results. The idea, thus, is not penalizing basic simulations with extra time.

Despite the above characterization, there exist simulators like Macedon [13] that are designed in a different way. Macedon provides an infrastructure to ease development, evaluation, and iterative design of overlay algorithms. Applications are built using a C-like scripting language, and code is automatically generated for TCP/IP and NS-2 (packet-level) simulator. But Macedon is mainly related to Domain-specific languages (DSLs) that generate functional code from domain specific representations. Besides, Macedon currently supports only two types of overlays: distributed hash tables and application level multicast. DSLs like Macedon are not designed to be extensible but instead to provide all possible functionalities and vocabularies in the domain language.

In consequence, we propose PlanetSim [10], an object-oriented, extensible, customizable, efficient framework for overlay network and services simulations implemented in Java programming language. In particular, we see these points as the main contributions of this paper:

• PlanetSim provides a clearly layered simulation framework, where researchers can easily develop their own protocols and services, simulating them in a time-efficient way. To do so, PlanetSim does not consider packet-level details.

• PlanetSim provides two ways of implementing new overlay protocols: an *algorithm-based* and a *behavior-based* approaches. We see the former as a traditional way to implement the overlay protocol itself all together by means of the node API. The latter approach enables the researcher to split every simple action a node must perform into *behaviors*, defining when such behaviors are applicable.

• As PlanetSim layered structure obeys to well-known pattern designs in software engineering, we provide a framework with clear hotspots (i.e., extension points), so that modifications and extensions to PlanetSim *at all levels* are easy and well defined. In particular, we introduce in Section 4 four useful extensions.

• From the one hand, PlanetSim defines by default a naïve mechanism to gather results, avoiding complex mechanisms that may slow down simulations. From the other hand, we have defined an introspection scheme, so that researchers can gather as much statistical information as necessary.

This paper is structured as follows. Section 2 introduces some background and Section 3 presents PlanetSim, and four of its useful extensions in Section 4. Finally, Section 5 concludes the paper.

2. BACKGROUND

Given that there exists a plethora of applications and services that can be build on top of overlays, one of the key challenges is to make the design and implementation of applications independent of the underlying overlay. To this end, the Common API (CAPI) [9] was proposed. The authors of CAPI define a layered design where services and applications can be built ones on top of others. The bottom tier is the so-called Key Based Routing (KBR) layer as the common denominator of services provided by structured overlays. Afterwards, on top of KBR layer, different services like distributed hash tables (DHTs), scalable multicast/anycast (CAST), as well as decentralized object location and routing (DOLR) can be defined. Eventually, specific services like Scribe [8] can be easily constructed, providing the whole set of services to end-user applications.

To do so, the CAPI provides two kinds of functions: the first ones for routing and processing messages in applications, *route* (downcall), *forward* and *deliver* (upcalls), and the second ones for accessing node's routing state information, *local_lookup*, *neighbourSet*, *replicaSet*, *range* (downcalls) and *update* (upcall).

3. PLANETSIM

In this section we describe PlanetSim's architecture, detailing in which hotpots our framework is extensible. Afterwards, in the following sections we point out some of its most interesting extensions, which demonstrate its true extensibility. Notice PlanetSim is freely downloadable under LGPL license from the website [6].

PlanetSim has been developed in Java language in order to reduce complexity and smooth the learning curve of our framework. We aim to create a framework that is easy to learn, easy to use, and easy to extend. The main drawback of this decision is the performance penalty that Java imposes. We however have carefully profiled and optimised the code to enable massive simulations in reasonable time. To validate the utility of our approach, we have implemented two overlays (Chord by the *algorithm-based* approach and Symphony by the *behavior-based* approach) and a variety of services like DHTs, CAST and DOLR. We have proved that PlanetSim reproduces the measures of these environments.

3.1 Design and Architecture

We believe simulator flexibility and extensibility is important in order to provide adaptability against different researchers requirements. To do so, we have defined PlanetSim as a layered architecture, where the different elements can be replaced easily to adapt the simulator correspondingly.

Firstly, we focus on the application and overlay interfaces to make applications and services implementation independent from specific overlays, enabling *application portability and code reusability* against different overlays. Thus, we have designed the interaction between services and overlays based on the CAPI described above. We argue this decision because CAPI provides such a unifying layer to different DHT systems, thus enabling to run the same application on

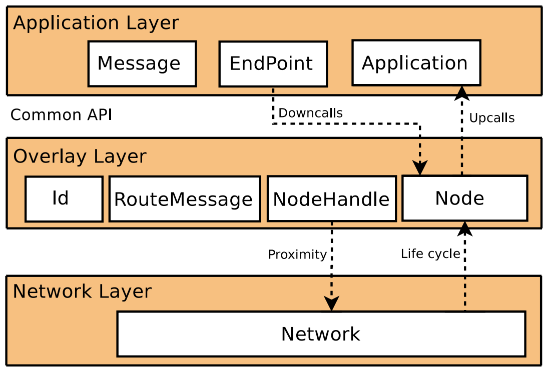

Figure 1: PlanetSim layered design.

top of different algorithms (e.g., Chord, Symphony, Pastry) as we expected.

Secondly, we describe in the following the layered architecture of PlanetSim, which allows its extensibility at all levels.

PlanetSim Layered Design

PlanetSim's architecture comprises three main extension layers constructed one atop another. As we can see in Fig. 1, overlay services are built in the top *application layer* using the standard CAPI facade [9]. This facade is built on the routing services provided by the underlying *overlay layer*. Besides, the *overlay layer* obtains proximity information to other nodes asking information to the lowest *network layer*.

The *network layer* dictates the overall life cycle of the framework by calling the appropriate methods in the overlay's Node and obtaining routing information to dispatch messages through the Network. As we will show later, the provided Network can be replaced for other ones in order to account latencies and load network models (e.g., GT-ITM, BRITE) to simulate more *realistic scenarios*.

PlanetSim can be extended at all levels. But for lack of space, we outline the three main extension points (hotspots) in our framework:

• **Application.** Developers of overlay services like Scribe must extend the Application class to implement the required messaging protocol. Application methods are upcalls from the overlay layer, which notify new messages. The Application code can then send or route messages using the EndPoint (downcalls) as well as access underlying node routing state. Any application created at this level can then be run or tested against any overlay in the next layer.

• **Node.** Developers of overlay algorithms like Chord must extend the Node class to implement the required overlay protocol. The Node provides incoming and outgoing message queues that permit to create the overlay infrastructure required in the upper layer. At this level, nodes exchange messages using Ids and NodeHandles (IP Address + Id).

• **Network.** It is possible to create customized Networks (CircularNetwork, RandomNetwork) by modifying specific configuration properties. In addition, one can construct its own network model taking into account proximity, so that the overlay structures become congruent with the underlying physical network.

Modularity and Extensibility

Into the development of PlanetSim, we have applied extensively different engineering techniques, like modularity, separation of concerns and design patterns in order to guarantee its modularity and extensibility. A simulation is mainly specified by a set of properties in plain text files. This way, we can modify widely a simulation only modifying these properties, that consist on class types and numeric values. In order to build all classes accordingly, we have defined a set of Factory classes following the Factory Method design pattern. For instance, a specific IdFactory implementation will build a specific Id implementation. To see why this is necessary, notice that Symphony's Ids consists of *float* numbers, while Chord's Ids are *natural* numbers of up 160 bits.

Another module into PlanetSim is that of Behaviors, that we detail in the following section. This module is responsible of the initialization of overlay Behaviors, their life cycle and the necessary software infrastructure to enable their use. The Results module is another key point into the simulator, because this provides a way to make outputs in different formats (currently GML and Pajek formats). This possibility is useful to interact with visual tools for protocol verification, for instance.

As a verification of the PlanetSim's true extensibility, we describe four altruistic extensions done by third parties in Section 4. And, given that developers are continuously interested in testing new and improved overlay algorithms, we outline in the following section the two ways overlay protocols can be implemented in PlanetSim.

3.2 Overlay Implementation Models

New overlay protocols can be tested into PlanetSim easily by implementing them based on two different ways, following the Algorithm Model or Behaviour Model. A Node implementation encapsulates all the algorithm from an overlay protocol. Following the life cycle, an overlay node *joins* the network, repeats the specific *stabilization* tasks (e.g., finding neighbours, setting long links, gossiping information, etc.) and finally *leaves* the network. Another non-expected operation is a node *failure*, that is also considered within PlanetSim. Both models are equivalent in the join, leave and optional fail operations, but they differ in the way nodes realize the stabilization task. Given that PlanetSim is a discrete event-based simulator, the stabilization procedure is mainly directed by the process of incoming messages that nodes send to another ones following the overlay protocol. Thus, intuitively, the Algorithm Model sets that the Node class itself is responsible of the message treatment. On the other hand, the Behavior Model is based on the separation of concerns. This way, the treatment of incoming messages is guided by the simulator, invoking the corresponding *Behavior* to process an incoming message. Basically, a Behavior encapsulates all the algorithm related to a single task from the overlay protocol. We detail both models in the following.

3.2.1 Algorithm Model

The node lifecycle into PlanetSim is dictated by the *join()*, *process()*, *leave()* or *fail()*. The join operation initializes the node in order to enter the network and start the communication with other nodes. Later on, the leave (or optionally failure) operation will be invoked to the node to leave the network. The difference between the leave and fail is that leave is a normal exit of the node from the network, performing any necessary communication with the neighbourhood, but a fail is an abrupt, unexpected exit of the node.

The main focus is then put into the *process()* method, responsible of the treatment of all incoming messages. Differentiating all types of messages, Nodes will process them accordingly to the overlay protocol. Usually this brings the developer to have lots of lines of code only for the message type detection and then the lines of code for the message process. This fact produces a protocol implementation that, depending on the abilities of the developer, could become difficult to maintain. The good point is that the protocol implementation is easy and direct. In Fig. 2 we can see the efficiency on simulating both Chord and Symphony, implemented using the Algorithm Model.

3.2.2 Behavior Model

To provide a greatest degree of *reusability*, PlanetSim includes an alternative model to encapsulate the actions a node performs in response to events. This model relies on the notion of *behavior*. In strict terms, a *behavior* is a Java class, namely Behavior, that specifies the action a node must perform in response to a specific kind of message. By a specific message we mean a message whose performative matches the *behavior's descriptor*. In essence, a behavior descriptor can be visualized as an expression that establishes when a Behavior must be executed. Specifically, a behavior descriptor is the result of the union of several fields, which attend to distinct reasons: The **message's type** represents a task, common between all nodes in the network; A **probability** adds uncertainty in the execution of Behaviors, useful to model Byzantine behavior for nodes; The **scope** refers to the message's recipient, allowing the values *LOCAL*, *REMOTE* and *ALWAYS* for when the message's recipient is the current node, any other or it must be ignored, respectively. Lastly, the **role** specifies that the Behavior must be executed for nodes that follow the protocol (*GOOD*) or not (*BAD*). If this field takes the *NEUTRAL* value means that this field must be ignored.

The key idea behind our *behavior-oriented programming* model is to allow developers to encode the set of actions any node can perform in separate classes that can be added and removed at will, without recompiling the source code to specify the way in which nodes must behave in each simulation. As an example, consider that a user wishes to test the fault-tolerance of a routing protocol in front of Byzantine failures. Using this model, we could define a set of Behaviors, one for each type of Byzantine failure, and add/remove them in each simulation to observe how the routing protocol behaves in front of specific permutations of failures. It is important to signal that the addition/removal of behaviors is done by merely modifying the configuration file. Next, we explain what happens at runtime when the simulator uses Behaviors to model nodes' actions.

Runtime Execution

At the heart of our model there is a singleton object called *behavior's pool* (BhP). The BhP keeps the instances of the Behaviors (i.e., one instance per Behavior), and acts as a *proxy* executing the corresponding behaviors on the nodes that have received new messages at the current step. To better understand this, we provide a simple example. Consider that a structured peer-to-peer overlay wants to replicate the contents stored under a key whenever a *Replicate* message arrives at a node. Using the original interface, a PlanetSim user would probably implement this operation modifying the *process()* method inside the node. However, in this approach, the user would implement it in a new Behavior, we call it *ReplicateBehavior*, avoiding the recompilation of the source code. In this case, once the programmer would have finished the implementation of the *ReplicateBehavior*, it would edit the configuration file to include the new behavior entry specifying to execute the ReplicateBehavior when a Replicate message arrives at a node.

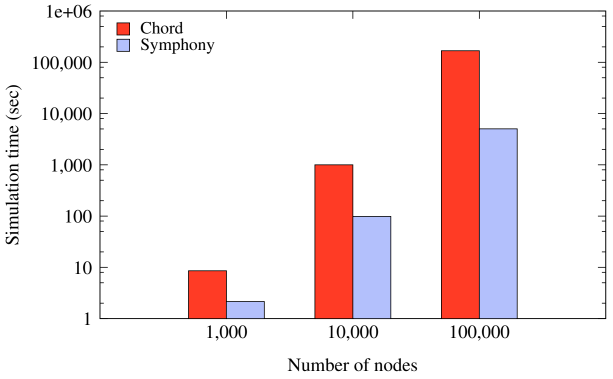

Figure 2: Simulation times.

The simulation proceeds as follows. At the start up, the simulator instantiates the BhP, and then, loads all the Behaviors specified in the configuration file. BhP intercepts the incoming messages and compares their performatives against the behavior descriptor, building a stack of Behaviors per message. Each stack contains the Behavior instances ordered from more specific to more generic, to provide an uncertainty model, which we believe could be useful to model Byzantine behaviors. Once a Behavior execution finishes, the BhP returns either the control to the node or spawns a new Behavior; this depends upon whether the stack of Behaviors has been completely executed or not.

It is important to note here that the implementation of the BhP is not fixed and an expert PlanetSim user can customize a new one to meet its own interests. For that purpose, the simulator includes several interfaces (e.g. BehaviorsFactory, BehaviorsPool, ...) to let developers customize the runtime behavior classes.

The unique shortcoming of this model is that Behaviors are also *singleton* objects and hence, they require the instance of the node to be passed as parameter. This slows down simulations, albeit not significantly (see Table 1). Nevertheless, we believe that for *many* applications (such the analysis of a routing protocol in front of Byzantine failures) the loss in scalability is by far compensated by the greatest flexibility of this approach. We provide the Symphony protocol implemented by means of Behaviors.

4. EXTENSIONS

In this section we detail some of the most interesting extensions made on PlanetSim by third parties, proving its flexibility and adaptability. We firstly focus on the *latency*

Table 1: Evaluation of Behaviors performance against simple TrivialP2P **protocol included in PlanetSim. Time is measured in seconds.**

Network Size	Using Behaviors	Lookup Time	Total Time	Incr. Ratio
100 Nodes	Yes	2.08	2.19	0.2327
	No	1.69	1.78	
1000 Nodes	Yes	248.24	252.27	0.2711
	No	195.29	196.84	

awareness extension of PlanetSim.

Researchers not only are interested in accounting number of hops for different simulation settings, but also accounting other network properties like the latency. In this way, we have extended Planetsim to add latency awareness in order to evaluate protocols in a more realistic context and more complex applications like content distribution networks. This extension is composed mainly of two components: a) a *parser* retrieves latency information from a network model and b) the *overlay routing protocol* is slightly modified to provide latency aware routing.

In first place, we define the *parser* hotspot to load into a *latency model* the communication cost information, which is usually represented by a graph file. Currently, we provide a Pajek graph file parser that computes all communication costs between peers during the initialization stage. For doing so, we define a *routing selection algorithm* to find the *best* route. In our case, we apply the Dijkstra's shortest path algorithm. However, other parsers and routing selection algorithms can be easily implemented (e.g., a GT-ITM file parser) following their interfaces. We have test it in a Chord similar to [16], and Fig. 3 shows some results.

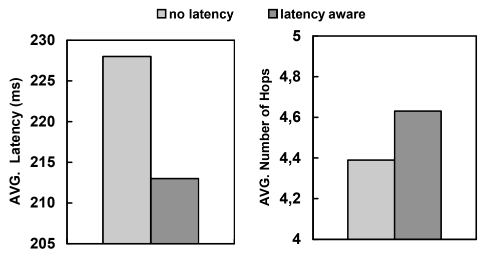

Figure 3: Latency evaluation on a Chord network of 500 nodes.

Other interesting extensions are i) how can the simulator extract statistics information accordingly, ii) how researchers can simulate various overlays within a single simulation and iii) how simulations can be time-improved by leveraging a multi-processor computer. As PlanetSim is focused on the efficient simulation of peer-to-peer networks rather than packet-level simulations, we believe that they are of great interest for the community. To see a detailed explanation, please refer to [12].

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduced PlanetSim, an extensible, customizable and efficient simulation tool for overlay and services simulation. The simulator is implemented in Java language under LGPL license and all work presented in this paper is fully downloadable. We detailed the simulator architecture, the two models to facilitate the overlay implementation and verification, such as our novel *behavior model*, as well as simulator's hotspots and extensions which prove the flexibility, adaptability and ease of use of PlanetSim.

PlanetSim has a live community, including users and developers, which has enabled to develop the extensions presented in this paper, as well as improve its performance and solve bugs. Now on, we are going to analyse how to provide a better PlanetSim performance and to design new abstractions to provide new functionalities within the simulator.

6. ACKNOWLEDGEMENTS

To Gerard París-Aixalà (Universitat Rovira i Virgili, email: gerard.paris@urv.cat) and Max Bruchmann (Technical University of Darmstadt, email: max-bruchmann@gmx.net) for their collaboration.

7. REFERENCES

[1] Gnutella. `http://www.gnutelliums.com`.

[2] OMNET++. `http://www.omnetpp.org/`.

[3] p2psim. `http://pdos.csail.mit.edu/p2psim/`.

[4] PeerSim. `http://peersim.sourceforge.net`.

[5] PlanetLab. `http://www.planetlab.org`.

[6] PlanetSim Website. `http://www.planetsim.net`.

[7] The Network Simulator (NS-2). `http://www.isi.edu/nsnam/ns/`.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8):1489–1499, 2002.

[9] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and S. I. Towards a Common API for Structured Peer-to -Peer Overlays. In *Proc. IPTPS'03*, February 2003.

[10] P. García, C. Pairot, R. Mondéjar, J. Pujol, H. Tejedor, and R. Rallo. PlanetSim: A New Overlay Network Simulation Framework. In *Proc. SEM'04*, pages 123–136, September 2004.

[11] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *ACM SIGCOMM Comp. Comm. Review*, 37(2), April 2007.

[12] J. Pujol Ahulló *et al.* PlanetSim: An extensible framework for overlay network and services simulations. Technical Report DEIM-RR-08-002, Universitat Rovira i Virgili, January 11 2008.

[13] A. Rodriguez, C. Killian, and S. Bhat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. NSDI'04*, March 2004.

[14] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218, pages 329–350, November 2001.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM '01*, pages 149–160, 2001.

[16] H. Zhang, A. Goel, and R. Govindan. Incrementally improving lookup latency in distributed hash table systems. In *Proc. SIGMETRICS '03*.