PlanetSim: An extensible framework for overlay network and services simulations

Jordi Pujol Ahulló[‡]
jordi.pujol@urv.cat

Pedro García López[‡]
pedro.garcia@urv.cat

Marc Sànchez Artigas[‡]
marc.sanchez@urv.cat

Marcel Arrufat Arias[‡]
marcel.arrufat@urv.cat

Gerard París Aixalà[‡]
gerard.paris@urv.cat

Max Bruchmann[*§]
max-bruchmann@gmx.net

[‡]Universitat Rovira i Virgili
Av. Països Catalans, 26
43007 - Tarragona, Spain

[§]Technical University of Darmstadt
Karolinenplatz 5
64289 - Darmstadt, Germany

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 precondictions. 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 for overlay networks and services, as well as extensions from third parties that prove its true extensibility and adaptability to the reserarchers' needs. In addition, we introduce within PlanetSim a novel way of 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, Experimentation

Keywords

Overlay network simulator, discrete event-based simulation, frameworks, design patterns.

1. INTRODUCTION

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 [11], and *simulated evaluation* where protocols are analyzed with some network settings assumptions. As a consequence of node unstability within those testbeds, lots of simulators have been appearing to help the research community, becoming standard platforms where different works are analized. Nevertheless, the vast majority of them are adhoc customized simulators [22] 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 simulation frameworks. In particular, we focus on structured (e.g. Chord [26], Pastry [25]) and unstructured (e.g. Gnutella [4]) peer-to-peer systems and services simulation onto our simulator PlanetSim.

In this way, we believe that the following points are the challenges to deal with in order to develop such an appropiate simulator.

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 simulations have a true high cost in time and computer resources, but high-level simulations show a better performance enabling, thus, big scale network evaluations. In addition, notice peer-to-peer researchers are usually more interested in algorithm verification than in simulating the whole 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 [17]) on top of them, regardless the overlay protocol. It is even more important because the simulator can provide lots of functions and researchers do not need to know them at the whole.

Modular, customizable and extensible. While the overlay research field is huge, such a simulator cannot be restrec-

[*]Working at QuaP2P Research Group.

tive on its provided functionality. Instead, we believe 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.

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 gathering 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 considerable amount of information. We believe that such a simulator should be flexible and provide both kinds of simulation results. The idea is not penalizing basic simulations with extra time.

To address these challenges, we propose PlanetSim [20], 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:

1. 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.

2. 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 different *behaviors*, defining when such behaviors are applicable.

3. As PlanetSim layered structure obeys to well-known pattern designs in software engineering, we provide a framework with clear hotspots, so that modifications and extensions to PlanetSim at all levels are easy and well defined. In particular, we introduce in Section 4 3 useful extensions.

4. 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.

The whole paper is defined as follows. Section 2 takes an overview of the existing simulation tools. Section 3 introduces PlanetSim and three of its useful extensions in Section 4. Section 5 concludes the paper.

2. RELATED WORK

There is a lot of work in the networking simulation field. Unfortunately, most of them are specialized in some way, so they do not provide a general simulation framework for peer-to-peer systems. Thus, for the sake of conciseness, we only detail the simulators of special interest for us.

We distinguish between network and overlay simulators. *Network simulators* provide packet-level simulation of network protocols (TCP, UDP, IP, etc), even awareness of delays, bandwidth and effects of TCP flows, over realistic Internet topologies. But there is an inherent cost on accounting all these low-level concerns, leading to an impoverished scalability for big networks. Instead, *overlay simulators* are usually more interested in evaluating overlay algorithms and its routing behavior without even taking into account the underlying network layer. The excessive overhead and complexity of network simulators thus imposes an unnecessary burden to overlay evaluators and researchers. For this reason, PlanetSim omits the low level details of internet routing substrate, because overlay networks conceal the details and pay little attention on them. This simplifies the implementation and greatly improves the performance. Nonetheless, PlanetSim can be easily extended, in particular, to load network models as we detail in Section 4.2.

Network simulators. There exist various low-level simulators like NS-2 [14], OMNET++ [8], J-Sim [6] and Narses [15]. For example, NS-2 [14] and OMNET++ [8] provide a standard framework for accurate simulation of network protocols. They are appropriate to simulate networks in the link, switching and transport layer. Besides, for smaller scale scenarios NS-2 and OMNET++ perform gracefully, but for overlays over a hundred in size suffer considerable scaling problems.

Overlay simulators. Many research groups have created their own overlay simulators, sacrificing accuracy for scale. Examples of these include PeerSim, FreePasty, SimPastry, 3LS, PLP2P, SimP^2, CANSimulator, GnutellaSim, NeuroGrid, OverSim, PeerfactSim.KOM, MAPLE, GPS, TOSim, ONSP and Query-Cycle Simulator. Some of them are for specific purposes and, thus, they are not for general peer-to-peer protocol evaluation. CANSimulator [2], FreeNet Simulator [23] and GPS (General Peer-to-peer Simulator) [30] only support CAN DHT, FreeNet and BitTorrent protocols, respectively. NeuroGrid Simulator [7] is focused on simulating searches over content distribution networks. is focused on simulating only protocol. Query-Cycle Simulator [18] is a cycle-based simulation framework for file-sharing peer-to-peer network simulation. TOSim (Trust Overlay Simulator) [27] is based on PeerSim [10] but emphasizing in trust and reputation integration within simulations. MAPLE [21] is a simulation system focused on nearest neighbor queries in mobile device environments. ONSP [28] is an overlay network discrete event simulation platform designed to run parallelized simulations using MPI. ONSP supports different network topology models and it is focused to simulate large overlay networks.

Some of them leverage low-level simulators. For example, OverSim [16] stands onto OMNET++ and has implemented differents underlays (INET, Simple and SingleHost) and overlays (Chord, Kademlia, Koorde and Broose) and demonstrates its feasibility to simulate large networks with the Simple underlay, that just avoids packet-level network routing simulation. As in PlanetSim, OverSim provides a Common API [19] to applications, thus making it very easy for developers to create and simulate complex distributed applications. Protocol specific details remain hidden from the

application-level point of view. Another example, GnutellaSim [5], runs on top of other network simulators (like NS-2). Nonetheless, the interface between node and application layers is *peer-to-peer specific*, making difficult reuse application implementations and do not scale to thousands of nodes. Thus, these approaches are absolutely tied to the packet-level simulator, making difficult its extensibility and scaling.

In the field of structured overlays, one of the pioneers is MIT's p2psim [9]. This simulator currently supports many protocols, including Chord, Accordion, Koorde, Kelips, Tapestry and Kademlia. p2psim is able to load network topology models (GT-ITM, etc) and it is protocol extensible, being pretty straightforward to develop new protocols by simply implementing the *join()* and *lookup()* low-level methods. Despite its protocol independence, p2psim provides no interface in order to simulate higher level applications. Besides, from the software engineering perspective, this simulator is poorly documented and difficult to extend for different purposes.

PeerSim [10] is a Java open-source component-based simulator which operates in two mode: event-based and cycle-based. PeerSim has good documentation and promises being flexible and, for instance, enables to define a stack of protocols on each node, with pluggable components. Cycle-based simulation promises to scale to big networks while event-based one permits more realistic simulation. We here focus on project activity and extensibility by third parties as a measure of whether a simulator is interesting and attractive: while people from around the world collaborate with PlanetSim and we show some of their extensions in this paper, the same thing does not occurs to other overlay simulators and, in particular, to PeerSim.

FreePastry [25], the Java open-source implementation of Pastry structured peer-to-peer protocol includes, as well, the possibility to simulate applications on top of this overlay network. As in PlanetSim, FreePastry provides a Common API [19] to the applications built on top of it. However, FreePastry is highly tied to the Pastry protocol, and it does not permit simulation of its applications on top of other structured peer-to-peer protocols.

Another interesting approach is the one followed by MACEDON [24]. 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. Moreover, it follows a standard API which does not tie applications to any specific overlay network protocol. Large-scale emulation and evaluation tools are at the developer's disposal as well. Macedon is not limited to structured P2P networks, and it includes an impressive variety of protocols and applications such as AMMO, Bullet, Chord, NICE, Overcast, Pastry, Scribe, and SplitStream. Furthermore, MACEDON simplifies development of new overlays using a finite state machine (FSM) model for defining overlay protocols.

MACEDON is a very nice tool for overlay simulation but it follows a completely different approach than PlanetSim. 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. In contrast, we have created a layered and modular framework that is *extensible at all levels*, and that can even be integrated with other frameworks. DSLs like MACEDON are not designed to be extensible but instead to provide all possible functionalities and vocabularies in the domain language.

3. PLANETSIM

In this section we describe PlanetSim's architecture, detailing in which hotpots (i.e. extension points) 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 [12].

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 distributed hash tables (DHTs), scalable multicast/anycast (CAST), as well as descentralized object location and routing (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. First, we focus on the application and overlay interfaces that enables running the same applications on top of different overlays. Afterwards, we detail our layered architecture design.

3.1.1 A Common API for Overlays

We aim to make independent applications and services implementation from specific overlays, to enable *application portability and code reusability* against different overlays. Thus, we have designed the interaction between services and overlays based on the Common API (CAPI) [19]. The motivation of this decision is the plethora of applications and services that can be build on top of overlays. We can see the CAPI diagram in Fig. 1.

In [19] authors 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 DHT, CAST and DOLR can be defined. Eventually, specific services like Scribe [17] can be easily constructed, providing the whole set of services to end-user applications.

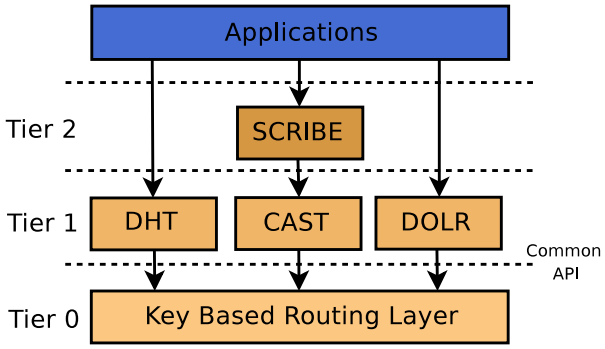

Figure 1: Common API diagram.

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).

We believe that CAPI provides such a unifying layer to different DHT systems, thus enabling to run the same application on top of different algorithms (e.g. Chord, Symphony, Pastry) as we expected. The API is, however, loosely defined and each research group is implementing its own version. This clearly hinders application interoperability and it only helps to improve understanding of applications in different DHTs through a common vocabulary. After evaluating different overlay architectures, we concluded that FreePastry is the cleanest and more advanced implementation of a structured overlay, with several applications implemented on top of it by means of a clean object-oriented implementation of the CAPI, but supporting only Pastry as overlay algorithm. We have decided, thus, to embrace FreePastry's CAPI implementation in our framework to leverage their existing code base and developers.

3.1.2 PlanetSim Layered Design

PlanetSim's architecture comprises three main extension layers constructed one atop another. As we can see in Fig. 2, overlay services are built in the *application layer* using the standard Common API facade. 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 *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*.

We outline 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.

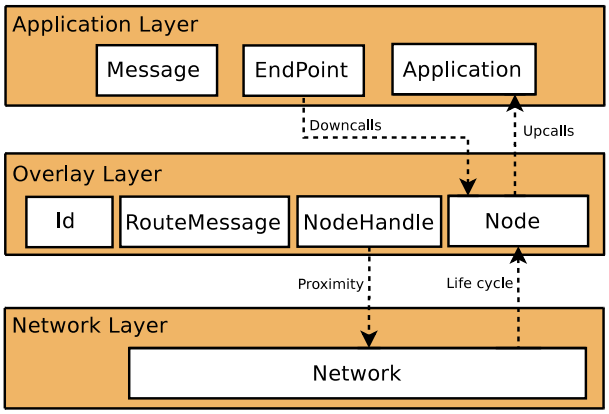

Figure 2: PlanetSim layered design.

- **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.

As a direct consequence of this layered approach, we can identify two user roles: the users interested in *overlay services* and the users focused on *overlay infrastructures*. The former can thus develop and test different overlay services on top of different overlay schemes. The other kind of users can be mainly interested in overlay analysis, from where the simulator provides the mechanisms to probe and compare different overlays, as well as to evaluate an overlay scheme against different network topologies (e.g. GT-ITM, BRITE).

Application Layer

At this layer we have followed FreePastry's implementation of the Common API. In this line, the interfaces borrowed from FreePastry are Application, EndPoint, Message, RouteMessage, Id and NodeHandle. We can see that this API is a facade to the underlying routing system of the simulator. This layer can thus permit very easily to test applications like DHT or Scribe multicast over different implemented overlays like Chord or Symphony.

We outline the Application and EndPoint classes as the main implementers of CAPI, where Application have the CAPI upcalls functions and the EndPoint includes the CAPI downcalls. In this way, Node notifies messages to Application by means of mainly *forward*, *deliver* and *update*. The EndPoint

is a facade to the underlying overlay Node and provides the *route* method and routing state methods like *replicaSet*.

Overlay Layer

The main conceptual entity and obvious hotspot of this layer is Node. A Node contains incoming and outgoing message queues and methods for sending and receiving/processing messages. Each particular Node must then define a complete behavior or protocol that will dictate which messages to send in specific times and how to react to incoming messages. Furthermore, to create a new overlay, the embedded protocol must define its own messages with specific information to arrange the overlay. This also implies that developers should be able to define their own message *types* and (a)periodic *tasks*.

As Fig. 2 depicts, the overlay layer have a bidirectional communication with application and network layers. Between application and overlay layers the CAPI communication exits. EndPoint and Node exchange RouteMessage entities, which define source, target and next hop information in order to route these messages accordingly. Between overlay and network layers, Node entity defines the *join*, *leave*, *fail* and *process* methods, which determine the life cycle at ovelay nodes. Specifically, *process* method has the necessary protocol each node maintains to create and maintain the overlay, while processing incoming messages and sending messages (if necessary) into the outgoing queue. Finally, Id entity enables to identify nodes in the overlay (and in particular in the simulator). Id must be implemented accordingly for each overlay protocol (e.g. we have defined Chord Ids by number types from 32 to 160 bits), while the IdFactory entity provides the specific way to build new Ids. Therefore, for implementing unstructured overlays like Gnutella into PlanetSim one only has to build in addition some unique key (e.g. unique integer values) as node Ids.

Network Layer

This layer is the main actor who dictates the overall life cycle. The simulator will run n simulation steps or until a specific goal (i.e. the network is *stabilized*) is achieved, while processing events. Events are node joins, leaves, fails or lookups, that can occur in different steps. In each step, the simulator moves outgoing messages to incoming queues for all nodes accordingly, and then calls the *process* method in each node to react to incoming messages.

The key hotspot is the Network: it represents the underlying network that the simulator uses to route messages. Thus, we could implement a GT-ITM topology in a Network in order to provide more realistic information like costs and latencies. Nevertheless, we provide a simple Network implementation without latency costs, in order to focus on algorithm verification, without proximity information worsening the simulator performance.

Another key possibility is to integrate our simulator with network simulators (e.g. NS-2, OMNET++). To do so, an appropiate Network and NetworkFactory will theoretically perform such an integration. This design opens new possibilities to feedback the overlay network with dynamic latencies or even node mobility for Mobile and Ad-Hoc Networks (MANETs). For example, a C++ implementation of

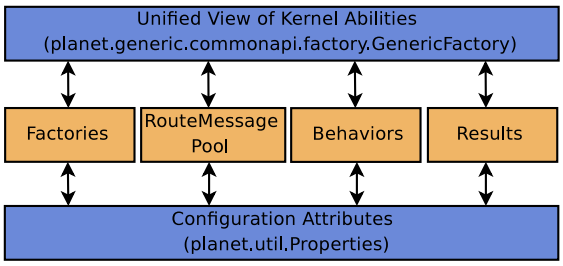

Figure 3: PlanetSim kernel diagram.

PlanetSim could provide the integration with NS-2 or OMNET++. In the same way, one could develop a Network implementation in order to run already implemented overlays in an experimental (TCP/IP) environment.

Another interesting feature of the simulator is to serialize to a file the full state of a simulation. This can be used for example, to stabilize a huge overlay network, serialize it, and later on resume the simulation from that point. This feature is extremely useful for large simulations and saves valuable computing time.

3.2 PlanetSim Kernel

We outline the amount of work done in PlanetSim kernel with thousands of Java source code lines and with good documentation in source code making it readable. Fig. 3 depicts the simulator kernel. The most visible entity within the kernel is the GenericFactory, which unifies all abilities the simulator provides. We detail all kernel's entities as follows:

- **Configuration Attributes.** We defined a Properties class to load, as well as to verify the correctness of the value types, and to provide within the simulator the whole specific current simulation configuration.
- **Factories.** Following the Factory Method design pattern, we defined a set of Factory classes to build accordingly all class types. For example, a specific IdFactory implementation would build a specific Id implementation. Notice that Symphony's Ids are *float* numbers, but Chord's Ids are *natural* numbers of up 160 bits.
- **RouteMessage Pool.** For every discrete simulation step overlay nodes process lots of messages (RouteMessages) from their incoming queue and to the outgoing queue. As this step is actually very intensive, we have designed this pool to reuse them as much as possible once they are free. This prevents then the Java Garbage Collector execution, saving notably time of simulation. This pool has an API to easily get RouteMessages instances correctly initialized. Fig. 4 shows the default RouteMessage diagram. Additionally to routing information (*source*, *nexthop* and *target*), it defines a *type* and a *mode* to help differentiating all kinds of communications into an overlay protocol.

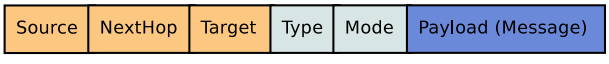

Figure 4: RouteMessage diagram.

- **Behaviors.** We add within the simulator a novel way to implement new overlay protocols. Briefy, a behavior is a Java class that encapsulates a simple action a node

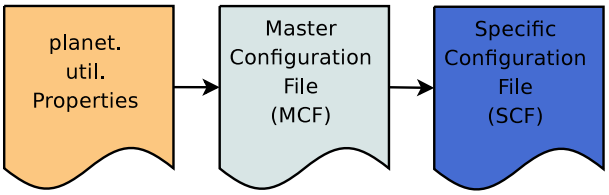

Figure 5: Configuration loading process.

must perform in response to certain event(s). Thus, an overlay protocol can be defined completely by a set of Behaviors. We detail Behaviors in the following section.

- **Results.** We provide a way to make outputs in different formats, detailing the overlay network setting. We provide GML and Pajek result formats. This extension point helps to employ visual tools which are very useful for the protocol verification.

- **Unified View of Kernel Abilities.** There exist a great number of class instances to use simultaneously to complete a simulation. Following the Composite design pattern, we have defined the GenericFactory as a facade to all functionality provided by several instances needed during a simulation. This class has the advantages of hiding instances interdependences and improving the memory usage by maintaining only one instance for each class type.

We provide a way to save notoriously time for repetitive simulations while verifying protocols, by means of the configuration loading process. As depicted in Fig. 5, Properties class loads firstly the Main Configuration File (MCF). MCF contains one or more lines[1] of the form ***TEST_NAME = specific_file*** for every test (i.e. main application), which define the Specific Configuration File (SCF) with all expected configuration properties. SCFs contain differentiated blocks where all simulator parts are accordingly configured. Since simulator source code only retains the MCF filename, we provide easily a way for repeating simulations without recompilation and, thus, saving the developer time and from annoying repetitive tasks. Notice that this loading process is only executed once at the beginning during the simulator initialization.

3.3 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, i.e. the message's *type* and *mode*, match the *behavior's descriptor*. In essence, a *behavior descriptor* can be visualized as an expression that establishes when a Behavior must be executed. In its primary form, a *behavior descriptor* can be a pair of literals establishing the message's *type* and *mode*. However, it must be noted here that *behavior descriptors* can be more complex. Later, we will delve into this.

[1]If there are various lines, exactly one line appears only uncommented.

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 our model, we could define a set of Behaviors, one for each type of Byzantine failure, and add and 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.

3.3.1 Runtime Execution

Thus far, we have introduced the notion of *behavior*, but we have not explained what happens at runtime when the simulator uses Behaviors to model nodes' actions. Next, we describe this.

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 *dispatcher* 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 the ReplicateBehavior when a Replicate message arrives at a node.

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. At that point, the BhP is ready to invoke Behaviors. For doing so, it intercepts the incoming messages and compares their performatives against the *behavior descriptors*. An interesting feature of the current BhP implementation is that any message can match more than one *behavior descriptor* at the same time. The idea is that PlaneSim can reuse the same task for more than one message type. Note that many protocols tend to perform the same set of tasks for almost all types of message arrivals. So, PlanetSim should support multiple behavior invocations per message. In order to do this, the BhP maintains a stack of Behaviors for every message. Furthermore, 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.

The BhP invokes a Behavior passing a pair of arguments. These are the original RouteMessage and the Node to which the RouteMessage is addressed. The reason behind passing the Node lies on the necessity of the *targeted* Node to update its internal state as a result of the invoked Behavior. In our example, it's clear that once the ReplicateBehavior

is executed on an arbitrary node, this node must replicate the requested key and update its internal state to maintain which of its neighbors are indeed storing the key.

Once the 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. In our example, after intercepting a Replicate message, the BhP will dispatch the ReplicateBehavior, and then, it will yield the control to the node.

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.

3.3.2 Behaviors Descriptors

A *behavior descriptor* is the result of the union of several fields, which attend to distinct reasons. Next, we describe these fields:

1. **Message's type.** The type represents a task. A task can involve a unique message, a query and response, or even a complex interaction. However, the basic idea is that, while being in the same task, any message maintains the same type.
2. **Message's mode.** It stems from the fact that in any task we require to identify in which state we are.
3. **Probability.** This property let the BhP add uncertainty in the execution of Behaviors. In general, this field can be useful to model a Byzantine behavior for nodes.
4. **Scope.** The scope property refers to the message's recipient. It can take three literals, which are:
   (a) ***LOCAL.*** This literal indicates to the BhP that the Behavior will be executed only if the message is destined for this node.
   (b) ***REMOTE.*** It tells the BhP that the current Behavior will be executed only if the recipient is a *remote* node, distinct from this one. This option has been devised to write down routing protocols in behavioral form. That is, we need that a *RoutingBehavior* executes at each hop unless the current node is the destination.
   (c) ***ALWAYS.*** It establishes that the value for this property must be ignored.
5. **Role.** It tells the BhP that the behavior execution can be conditioned on the node's *role*. From a protocol's viewpoint, a node is ***GOOD*** when it follows the protocol; and ***BAD*** otherwise. The literal ***NEUTRAL*** forces the BhP to ignore the role of the Behavior. Notice that this property provides a natural mechanism to examine the fault-tolerance and security issues in peer-to-peer protocols. Developers can write two sets of disjoint Behaviors, one set for the ***GOOD*** nodes, and one for the ***BAD*** nodes to investigate what happens when a specific fraction of nodes does not behave properly.

The *type* and the *mode* depend on the kind of messages required by the peer-to-peer protocol to implement. They can take literal values like ***DATA*** or ***REQUEST***. Nonetheless, they can also take a pair of powerful wildcards: the '?' (i.e. the *complementary* wildcard) and the '*' (i.e. the *universal* wildcard). The universal wildcard permits a Behavior to be executed irrespective of the current value of the *type* and *mode* of the incoming message. In contrast, the complementary wildcard forces the BhP to execute a Behavior unless there is a most specific combination of message's *type* and *mode*. For example, suppose there exists a Behavior A for the pair $< DATA, REQUEST >$ (we treat the union of the message's *type* and *mode* as a pair: $< type, mode >$). In addition, consider that the overlay uses 3 message modes ***REQUEST***, ***REPLY***, and ***REFRESH***. Then, associating the pair $< DATA, ? >$ to a Behavior B, $B \neq A$, has the consequence that any message with the performatives $< DATA, REFRESH >$, and $< DATA, REPLY >$ causes B to be invoked instead of A.

As aforementioned, the BhP maintains for each message a stack of Behaviors ordered from more specific to more generic. The precedence is listed on following table with the pairs (i.e. $< type, mode >$) ordered from the most specific to the most generic:

Type	Mode	
Literal	Literal	+ spec.
Literal	?	
Literal	*	
?	Literal	
*	Literal	
?	*	
*	?	
*	*	- spec.

As an example, we include a configuration file with 3 Behaviors to illustrate a simple specification (configuration file):

RoutingBehavior	?	*	1.0	REMOTE	NEUTRAL
JoinBehavior	JOIN	REQUEST	1.0	LOCAL	NEUTRAL
Databehavior	DATA	*	1.0	ALWAYS	NEUTRAL

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. EXTENSIBILITY

In this section we detail some of the most interesting extensions made on PlanetSim by third parties, proving its flexibility and adaptability. Specifically, we detail i) how can

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
	Yes	195.29	196.84	

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 these features are of great interest for the community.

4.1 Gathering and Showing Information

Another way for adding more functionality to software is aspect orientated programming (AOP). AOP is really interesting because when disabled it penalizes neither in terms of computation time nor memory usage of a expected time-efficient simulation. We use this approach to extend simulator functionality with a clear separation of concerns, for example for *statistics*. AOP could also provide the possibility to remove unused functionality from PlanetSim. For implementing aspects we use AspectJ [1] that works very smoothly with the Java programming language.

4.1.1 Gathering Information

In our new branch we have build an aspect to gather statistics and write them in a Gnuplot [3] readable file format. However, one can build her own aspects to gather any detailed information from the simulation. By using the PlanetSim statistics aspect, the simulator creates after each simulation run several data files (.dat) and a control file (.plt) which can be executed by Gnuplot. This statistics aspect gathers information about the amount of messages that are sent in each step of the simulation. As messages differ in their *type*, *mode* and *content*, the data files are named by the message type, mode, content and the name of the simulation. The control file's name is concatenated by the the word *traffic* and the name of the simulation, for example "**traffic.HELLOWORLD_DHTPEERTEST.plt**".

4.1.2 Showing Information

The simulator has a hotspot to export the simulated network status in different formats (currently available GML and Pajek). However, it could be interesting to integrate some existing tools with the simulator, in order to leverage their visualization abilities. Prefuse [13] is a visualization toolkit that lastly is having a lot of attention. Prefuse is implemented in Java under BSD licence, with clear APIs and good documentation. Even more, Prefuse can produce progressive visualization, thus a step-by-step visualization of the simulation run could be performed.

As a result of our statistics aspect, the simulator provides other ways to show results. In this case, we have decided to use Gnuplot for the results visualization, but one can decide to gather the information in other formats. Gnuplot is a tool to plot data and functions in various visualization forms. A common way to insert data into Gnuplot is to create data files with two columns for the x and the y values and a control file where is defined which data should be drawn. In case of a simulation that deals with a lot of different messages Gnuplot will plot a very complex diagram. For getting a more clear view it is recommended to comment out the unused curves in the control file by putting "#" in front of the specific line.

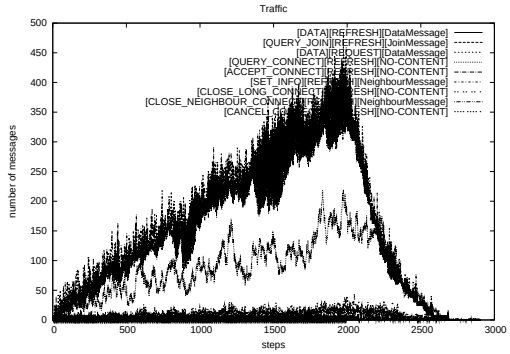

Figure 6: PlanetSim created statistics on "HELLOWORLD_DHTPEERTEST" test

4.1.3 Gnuplot Output Example

After running the "HELLOWORLD_DHTPEERTEST" on a Symphony overlay with 1000 nodes and a duration of 8.828 seconds, Gnuplot can now visualize the diagram depicted in Fig. 6. The x-axis indicates the simulation steps and the y-axis the amount of a messages at a step. Moreover, Gnuplot can also visualize diagrams in colors.

4.2 Latency-aware simulation

Researchers not only will be interested in accounting number of hops for different simulation settings, but also accounting other network properties like latencies. In this way, we have extended Planetsim to add latency awareness in order to evaluate in a more realistic context sensitive 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.

The other key point is that nodes route messages taking advantage of latency information provided by the *latency model*. But, this issue depends though highly on the overlay and needs of a specific implementation for each protocol: Chord, Symphony, etc. However, changes are localized into the Node class that implements such an overlay protocol.

We have modified our Chord in a similar way of that LPRS-Chord [31]. Despite Chord defines greedy routing to send messages to the furthest located nodes, using the latency model information nodes can now configure their finger tables by choosing such nodes belonging to each finger interval, but reducing the distance in terms of identifier space as well as latency. In consecuence, this mechanism clearly implies a *tradeoff between reducing latency and increasing the number of hops*, which can be balanced easily in the node implementation.

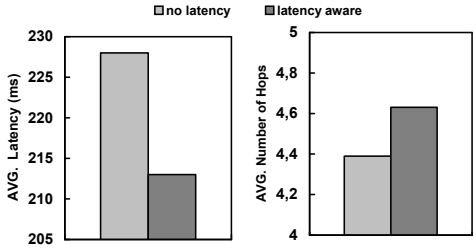

Figure 7: Latency evaluation on a Chord network of 500 nodes.

Fig. 7 depicts the evaluation of considering (or not) latency information for routing in a 500-node network of our modified Chord, where latency information was taken from the PlanetLab network to use a *realistic* model. We can see how the overall latency is reduced (left), whereas the total number of hops is almost the same (right). This methodology clearly improves routing efficiency and *enables further evaluations of network effects into the overlay behavior.*

4.3 Multi-network simulation

Real networks are usually composed by complex interconnected systems. For instance, Internet is formed by a set of autonomous systems (AS) that interact using an inter-domain protocol, whereas internal routing is provided by a different protocol. In overlay networks it is also possible to set up a system composed by several overlays that interact between them at the same time, where each overlay maintains its routing behavior. For example, hierarchical overlays [29] require of various overlay networks interconnected by routing-peers to compose a scalable system.

Considering the utility of multi-network simulations, we added to PlanetSim the support for multiple overlay networks. This feature allows creating *one, two or more overlay networks* in the same simulation run. To do so, we introduced the notion of the *superpeer node*, a special node that connects two or more overlays. Thus, *communication between applications is allowed even if they are running in different overlays.*

The configuration of multi-network simulations is also very flexible. The overlay networks used in the simulation can be configured separately, and they can present different options, since each overlay has its own configuration file. It is also possible to use multiple topologies, for instance, a simulation of a Chord overlay and two Symphony overlays interconnected. For doing so, we have added some configuration properties: the number of nodes that will act as superpeers, as well as which overlay networks will interconnect each superpeer.

Enabling inter-overlay communication is easy. The routed messages need the overlay identifier, an extra parameter indicating the destination overlay. Nodes perform inter-overlay routing by means of an entity with global knowledge of the location of superpeers and the multi-network structure, in the same way that nodes know how to contact to superpeer nodes in a hybrid architecture. This simplified routing mechanism provides the most suitable local superpeer and also the minimum distance in terms of superpeer hops between a source overlay and a destination overlay.

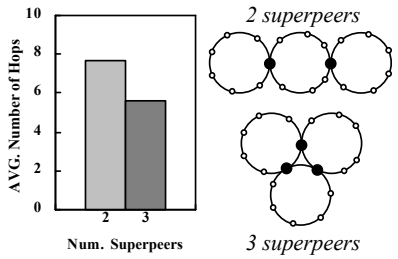

Figure 8: PlanetSim multi-network experiment.

As many other parts of PlanetSim, multi-network simulation provides several hotspots to extend the framework and adapt to different simulation environments. In this case, the main hotspots are the *superpeer selection mechanism* and the *routing policy* employed to select the suitable superpeer that must route the message up to the target overlay.

Several simulations were performed to test this extension, such as testing different overlay networks with various sizes and overlay interconnection topologies. Fig. 8 shows the results of a test accounting the mean number of hops of 100 messages sent to random destinations. The same multi-network configuration (two Chord overlays and one Symphony overlay, with 100 nodes each) was tested with 2 and 3 superpeers. Since 3 superpeers assure the interconnection between all 3 overlays, the mean hop count decreases with respect to the configuration with 2 superpeers.

4.4 Multi-thread simulation

The original PlanetSim release was designed to run on single CPU computers, but the current trend is using multi-core CPU machines. Therefore we created a new PlanetSim branch which focuses on supporting this technology. In the original PlanetSim the simulator processed in each simulation step sequentially over every node by calling the **boolean** process(**int**) method. This allows nodes locally to read messages from the incoming queue and to put new messages in the outgoing queue.

In the new version, in each simulation step the whole node set is split up in K groups and each group is now processed by an own thread. The number of groups K can be specified in the simulation's properties by assigning the number of groups K to the property ***SIMULATOR_PROCESSORS***. For example, if you have a cpu with two cores, you can use the following setting: ***SIMULATOR_PROCESSORS = 2***. In a simulation with for example 1000 Nodes the simulator distributes calling the method *process* among the two cpu cores. Thus the simulator needs less time to finish its work. Table 2 shows some results of this improvement. Notice that since parallelization is performed for every simulation step (not for the whole simulation process), the speedup is not quite high. Nevertheless, we believe that an improvement of around 30% on the simulation time with a dual core computer is by far notorious.

4.4.1 Programming restrictions for multi-threading simulations

As the nodes act locally, there are only a few things to consider. Behaviors should not have fields, because in the current version there is one single BehaviorPool. This means that there are only unique behaviors which are shared be-

Table 2: Evaluation of multi-threading.

Network Size	Num. Threads	Total Time (sec.)	Speedup
10000	1	480.266	
	2	367.344	1.30
1000	1	100.782	
	2	76.875	1.31

tween the processing threads. Another problem at the moment is that program calls like Results. incTraffic () or calls that retrieves messages (GenericFactory.getMessage()) from the MessagePool had to be declacered as synchronized which can lead to a bottleneck in the performance with multiple processors.

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduced PlanetSim, an extensible, customizable and efficient framework 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 from PlanetSim Website [12]. We detailed the simulator architecture, our novel *behavior model* to facilitate overlay implementation and verification, as well as simulator's hotspots and several extensions which prove the flexibility, adaptability and ease of use of PlanetSim.

PlanetSim has a live community, including users and developers from around the world, 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. ACKNOWLEDGMENTS

Authors appreciate feedback and work from contributors and developers and for believing in PlanetSim Project. This work has been partially funded by the European Union under the 6th Framework Program, POPEYE IST-2006-034241, and by the Spanish Education and Science Minitry, AP-2006-04166 FPU grant.

7. REFERENCES

[1] AspectJ. `http://www.eclipse.org/aspectj`.
[2] CANSimulator. `http://sourceforge.net/projects/cansimulator`.
[3] Gnuplot. `http://www.gnuplot.info`.
[4] Gnutella. `http://www.gnutelliums.com`.
[5] GnutellaSim. `http://www.cc.gatech.edu/computing/compass/gnutella/`.
[6] J-Sim. `http://www.j-sim.org/`.
[7] NeuroGrid. `http://www.neurogrid.net/`.
[8] OMNET++. `http://www.omnetpp.org/`.
[9] p2psim. `http://pdos.csail.mit.edu/p2psim/`.
[10] PeerSim. `http://peersim.sourceforge.net`.
[11] PlanetLab. `http://www.planetlab.org`.
[12] PlanetSim Website. `http://www.planetsim.net`.
[13] Prefuse. `http://www.prefuse.org`.
[14] The Network Simulator (NS-2). `http://www.isi.edu/nsnam/ns/`.
[15] M. Baker and T. Giuli. Narses: A scalable flow-based network simulator. Technical report, Stanford University, November 2002.
[16] I. Baumgart, B. Heep, and S. Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proc. GI'07*, pages 79–84, May 2007.
[17] 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.
[18] M. T. S. T. E. Condie and S. D. Kamvar. Simulating a File Sharing P2P Network. Technical report, Stanford University, 2002.
[19] 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.
[20] 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.
[21] W.-S. Ku, R. Zimmermann, C.-N. Wan, and H. Wang. MAPLE: A Mobile Scalable P2P Nearest Neighbor Query System for Location-based Services. In *Proc. ICDE'06*, page 160, 2006.
[22] 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.
[23] J. Pfeifer. Freenet Caching Algorithms Under High Load. `http://www.cs.usask.ca/classes/498/t1/898/W7/P2/freenet.pdf`.
[24] A. Rodriguez, C. Killian, and S. Bhat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. NSDI'04*, March 2004.
[25] 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.
[26] 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, New York, NY, USA, 2001. ACM Press.
[27] W. Wang and G. Zeng. A Generic Trust Overlay Simulator for P2P Networks. In *Proc. PRDC'06*, pages 401–402, December 2006.
[28] Y. Wu, M. Li, and W. Zheng. ONSP: Parallel Overlay Network Simulation Platform. In *Proc. PDPTA'04*, pages 1147–1153, June 2004.
[29] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proc. ICDE'03*, pages 49–60, March 2003.
[30] W. Yang and N. Abu-Ghazaleh. GPS: a general peer-to-peer simulator and its use for modeling BitTorrent. In *Proc. MASCOTS'05*, pages 425–432, September 2005.
[31] H. Zhang, A. Goel, and R. Govindan. Incrementally improving lookup latency in distributed hash table systems. In *Proc. SIGMETRICS '03*.