PlanetSim: A New Overlay Network Simulation Framework

Pedro García, Carles Pairot, Rubén Mondéjar, Jordi Pujol,
Helio Tejedor, and Robert Rallo

Department of Computer Science and Mathematics, Universitat Rovira i Virgili,
Avinguda dels Països Catalans 26, 43007 Tarragona, Spain
{pgarcia, cpairot, rrallo}@etse.urv.es

Abstract. Current research in peer to peer systems is lacking appropriate environments for simulation and experimentation of large scale overlay services. This has led to a plethora of custom made simulators that waste development resources and hinder fair comparisons between different approaches. In this paper we present a new simulation / experimentation framework for large scale overlay services with three main contributions: i) provide a unifying approach to simulation/ experimentation that eases the transition from simulation to network testbeds, ii) it clearly distinguish between the design of overlay algorithms (key based routing), and the applications and services built on top of them, iii) offer a layered and modular architecture with clear hotspots, and pervasive use of design patterns. We have used PlanetSim to implement and evaluate overlay networks such as Chord and Symphony, and overlay services such as Scribe application level multicast, and keyword query systems over distributed hash tables.

1 Introduction

In the last years, we have experienced an increasing interest in peer to peer systems from research settings but also from commercial vendors because of its mainstream use in the Internet. Furthermore, the growing bandwidth and computing power in the edges of the network foresee innovative massive applications of peer to peer technology.

We can classify peer to peer networks as structured or unstructured, depending on the way they are connected and how the data they contain is arranged. In a structured network the connections between nodes are of some regular structure, which allows deterministic and optimal lookup hops (typically O (log N)). In contrast to structured networks, nodes in unstructured networks do not share a regular structure and a unified identifier space. Lookups are thus normally achieved by flooding and using replication in the network.

Structured P2P networks are now a hot research topic and they represent an interesting platform for the construction of resilient, large-scale distributed systems. Moreover, structured networks can be used to construct services such as distributed hash tables (DHT), scalable group multicast/anycast (CAST) and decentralized object location and routing (DOLR). We focus our research in PlanetSim on structured overlays and the design and development of distributed services on top of them.

In general, both structured and unstructured networks are often called overlay networks because they are built on top of an existing network, usually on top of the Internet. At the moment, P2P networks usually do not map the underlying network or even do not take the layout of these networks into account. As we can see, these overlay networks are thus working at the application layer, and use transport protocols like TCP or UDP as communication channels between inter-connected peers.

P2P researchers are usually more interested in algorithm verification (number of hops, node stress, link stress) than in simulating the whole TCP/IP stack. As a direct consequence, researchers find existing network simulators too specific and low-level. Besides, those simulators exhibit a considerable lack of scalability for thousands of nodes. Another key problem is that the transition from simulated code to experimental code is still quite difficult to achieve.

This has led to the development of ad-hoc simulators (SimPastry, FreePastry, p2psim, DKS, Tapestry) from a high number of research groups, wasting expensive resources in infrastructure code and avoiding clean comparisons between different algorithms. With minor differences, all these ad-hoc simulators are poorly documented and do not show clear-cut software engineered designs. Due to these approaches it is quite difficult to reuse code and even harder to extend those simulators.

To address these limitations, we present PlanetSim, an object oriented simulation framework for overlay networks and services. The novel contributions of PlanetSim are the following:

1. PlanetSim presents a layered and modular architecture with well defined hotspots documented using classical design patterns. This can considerably reduce the learning curve and thus ease the development of new overlay services and algorithms.
2. PlanetSim clearly distinguishes between the creation and validation of overlay algorithms (Chord, Pastry) and the creation and testing of new services (DHT, CAST, DOLR) on top of existing overlays. Our layered approach cleanly decouples services built in the application layer using the standard Common API for structured overlays [2], and peer to peer algorithms built in the overlay layer.
3. PlanetSim also aims to enable a smooth transition from simulation code to experimentation code running in the Internet. Because of this, we provide wrapper code that takes care of network communication and permits us to run the same code in network testbeds such as PlanetLab. Furthermore, because we follow FreePastry's implementation of the Common API, our overlay services can easily run on top of Rice's FreePastry Java code. This enables complete transparency to services running either against the simulator or the network.

PlanetSim has been developed in the Java language to reduce complexity and smooth the learning curve in our framework. We however have profiled and optimised the code to enable scalable simulations in reasonable time. To validate the utility of our approach, we have implemented two overlays (Chord and Symphony) and a variety of services like CAST, DHT, and DOLR. We have proved that PlanetSim reproduces the measures of these environments and is also efficient in its network implementation.

This paper is organized as follows. Section 2 gives details of the PlanetSim framework architecture and services. We present the framework's validation using developed extensions in Section 3. Section 4 compares PlanetSim with related approaches, and finally we draw conclusions and present future work in Section 5.

2 PlanetSim Architecture

The overall model comprises a discrete event simulator (time-stepped) that uses a central step-clock to simulate timing. As we will explain in this section, most entities in an overlay simulator are related to the routing of messages between the nodes of the overlay. Nevertheless, overlay simulators must not forget the underlying network that sustains the overlay and thus include appropriate abstractions and mappings for both routing infrastructures.

We have decided to implement PlanetSim in Java in order to smooth the learning curve of the framework. We aim to create a framework that is easy to learn, easy to use, easy to extend, and easy to integrate with other frameworks. 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.

2.1 The Common API for Structured Overlays and FreePastry

To better understand the overall architecture we must first introduce the Common API for Structured Overlays and the FreePastry implementation. We propose a novel service to be supported by overlay simulators: a façade API to develop overlay services and applications on top of existing overlays. This API is based on the proposed Common API (CAPI) for structured Peer-to-Peer overlays published in [2]. The main motivation for this decision is the plethora of applications and services that can be built on top of structured overlays.

In [2] authors identify the Key based Routing (KBR) as the common denominator of services provided by any structured overlay. Every node in a structured overlay is thus responsible for a number of keys in the identifier space (key's root), and can route messages in O(log N) hops to the keys's root for any key.

On top of this Tier 0 KBR, structured overlays can be used to construct services like distributed hash tables, scalable group multicast/anycast and decentralized object location (see Figure 1). These services in turn promise to support novel kinds of distributed applications like notification systems, messaging,

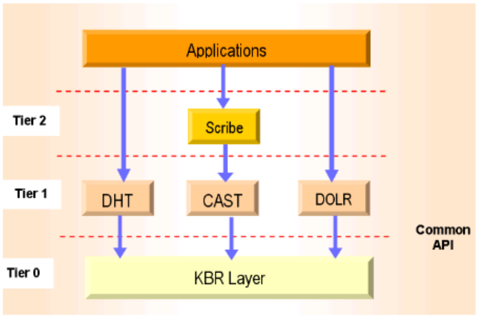


Fig. 1. Common API Diagram

content distribution networks and cooperative replication of archival storage. Furthermore, many traditional applications like Usenet or DNS have recently been re-architected on top of these decentralized architectures.

The common API offers two kinds of functions: the first ones for routing and processing messages in applications, and the second ones for accessing node's routing state information. The former include three kind of calls: *route*, *forward* and *deliver*. The *route* operation delivers a message to the key's root. Applications process messages by executing code in upcalls (*forward*, *deliver*) which are invoked by the underlying routing system. The *forward* upcall is invoked at each node that forwards a message and enables to override the default routing behaviour. The *deliver* upcall is invoked on the node that is root for a key upon the arrival of the message.

The second kind of functions for accessing node's routing state includes *local_ lookup*, *neighbourSet*, *replicaSet*, *update*, and *range*. We will not explain each function due to lack of space, but all of them give information about routing state and identifier space information from running nodes.

Using these functions, the authors in [2] define the mapping to different overlay algorithms, and they also specify how to construct overlay services like DHTs, CAST or DOLR.

The common API (CAPI) promises a unifying layer to different DHT architectures, and thus enabling to run applications on top of different algorithms (Chord, Pastry, Tapestry). 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 systems, we concluded that FreePastry is the cleanest and more advanced implementation of a structured overlay. They offer a clean object oriented implementation of the common API in the Java language. Besides, they have implemented several applications on top of this API like Scribe overlay multicast, replication systems like PAST and others. FreePastry is an active project and many research groups are using FreePastry code to create new innovative P2P services.

Nevertheless, FreePastry is also poorly documented and it is only extensible at the application level. It is not possible to implement and simulate other overlay algorithms apart from Pastry. Because of this, we have chosen to embrace FreePastry's common API implementation in our framework to leverage their existing code base and developers.

2.2 PlanetSim Layered Design

PlanetSim's architecture comprises three main extension layers constructed one atop another. As we can see in figure 2, overlay services are built in the application layer using the standard Common API façade. This façade is built on the routing services offered 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 explain later, the Network layer can be implemented either by the NetworkSimulator or NetworkWrapper. Developers can thus transition from simulation to experimentation environments in a transparent way.

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 underlying layer and they notify of specific 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 structured 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 KBR infrastructure required in the upper layer. At this level nodes interchange messages using Ids and NodeHandles (IP Address + Id).
- Network: It is also possible to create customized Networks (RingNetwork, CircularNetwork, RandomNetwork) by selecting specific Id Factories and also to provide additional routing or proximity costs to the overall routing infrastructure.

As a direct consequence of this layered approach we can also identify two main user roles: ones interested in overlay services and others focused on overlay infrastructures. The former can thus develop and test different overlay services on top of different KBR schemes or even probe services without even care about the KBR layer. Other kind of users can be mainly interested in structured overlays and thus use the simulator to probe or compare a variety of KBR algorithms.

For example, in our research group, there are researchers working at the application layer developing new replicated DHT services, and also experimenting

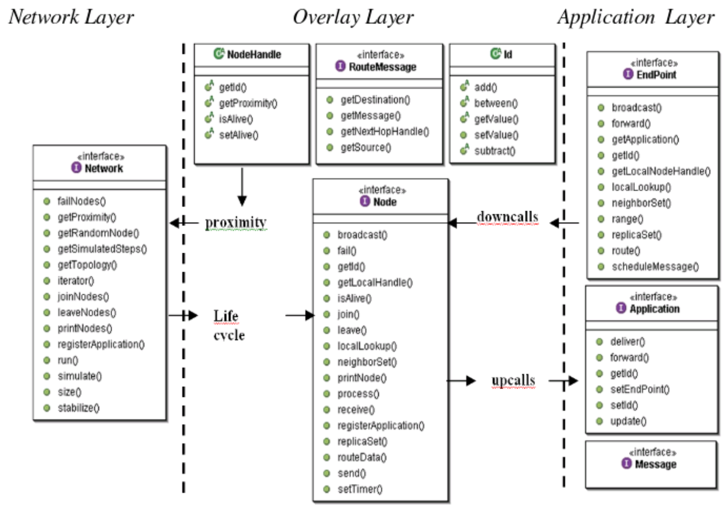


Fig. 2. PlanetSim class diagram

with query systems on top of different overlays. Another group is working at the overlay layer to compare security problems and solutions (BadNodes) over different overlays.

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 façade 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 the common API. The EndPoint is a façade to the underlying overlay Node and offers the *route* method and routing state methods like *replicaSet* or *range*. The Application is a hotspot containing the methods *deliver*, *forward* and *update* that will be invoked by the overlay layer accordingly on reception of messages. As we can observe, Application provides upcall messages invoked by the Node and EndPoint provides downcalls to access Node's routing state services.

In order to run an application (overlay service) in PlanetSim three configuration files are required: the simulator properties, the overlay properties, and the simulation properties.

To simulate an overlay we need to specify in those files a concrete Node (ChordNode) defining the overlay protocol, a concrete IdFactory (CircularIdFactory) and a specific Network (SimpleNetwork).

Each node includes a configuration file specifying different configuration parameters. For example, ChordNode file can define the number of bits in the identifier, stabilization period or other related parameters. Each Network can also be properly tuned defining its own parameters.

Finally, when a developer prepares an overlay simulation, he must define in a configuration file (overlay) several parameters like: Node Type (Chord, Symphony), Network Type, event file, log file, and others.

Configuration information is essential to accurately tune and probe new overlays or services, and to validate and compare existing results. The key concept here is that each hotspot includes its own configuration information file, and the final execution weaves the different components that create the running overlay testbed.

Overlay Layer

The main conceptual entity and obvious hotspot of this layer is the Node. A node contains incoming and outgoing message queues and methods for sending and receiving/processing messages. Each particular node must then include a complete behaviour 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.

At the overlay layer, the communication is bidirectional with both the application and network layers. With the application layer, the Node notifies the Application of received messages (upcalls) and it is invoked by the EndPoint façade in order to route messages or obtain routing state information (downcalls).

Both the EndPoints and the Nodes exchange RouteMessage types. A RouteMessage contains source and target identifiers, as well as information regarding the next hop in the overlay. It is also possible to modify the next hop route at the application or overlay layers in order to alter the routing scheme.

With the network layer, the Node hotspot provides the template methods (*join*, *leave*, *fail* and *process*) that determine the life's cycle of every node. The method *process* contains the specific protocol each node maintains to create the overlay. Besides, every node has an incoming and an outgoing message queue; incoming messages are parsed every step in the *process* method, and the *send* method moves messages to the outgoing queue.

To identify nodes in the overlay, the simulator employs three main entities: Id, IdFactory and NodeHandle. Ids are custom number types of 32 to 160 bits that identify nodes in the overall key based routing scheme. The extensible IdFactory permits to define custom Id generation schemes in each overlay. Additionally, NodeHandles contain IP to Id value pairs for each node. Furthermore, a NodeHandle provides a *proximity* method that queries the Network to obtain network proximity information.

As we can see, we have many upcalls that define the Node's life cycle and registering of applications, and only one downcall to query the Network for proximity between Nodes.

Network Layer

This layer is the main actor who dictates the overall life's cycle. The simulator will run n simulation steps or until a specific goal (i.e. the network is stabilized) is achieved. In each step, the simulator moves outgoing messages to incoming queues for all nodes, and then calls the *process* method in each node to react to incoming messages.

Furthermore, the simulator must process events in different steps. Events are node joins, leaves, fails, or lookups. Events can be generated from an event file declaratively, or programmatically using simulator APIs.

The key hotspot is the Network: it represents the underlying network that the Simulator uses to route messages. The Network contains a mapping of NodeHandles to Nodes that permit to correctly dispatch messages from source to destination.

An overlay can run on top of different networks using different underlying protocols. Developers can define their own networks, with specific protocols. The network can also include latency or cost information, or even the topology and arrangement of real nodes in this network. We could then implement a GT-ITM (Georgia Tech Internetwork Topology Models) transit stub topology in a network that would add more real information about costs and latencies.

Furthermore, each node can try to calculate its network proximity to other node. This can be defined in a NodeHandle's *proximity* method, transparently invoking the Network's *proximity* method (following FreePastry's interface definition). Developers can then decide in the network which proximity metric to employ (ping, landmarks, etc).

Nevertheless, a simple overlay mostly focused on algorithm verification, probably will be more interested in a very simple Network –without proximity information worsening the simulator performance–. In the current version of PlanetSim, we only provide simple Networks like RingNetwork, or CircularNetwork that do not include latency costs. It is however feasible to incorporate Peersim [8] or Brite [5] network information to define more realistic networks.

An ideal case at this point could be the integration of disparate frameworks: overlay frameworks with network simulation frameworks. The Network hotspot and Network factory extension point would theoretically permit to create such integration points. This is to say for example between J-sim and PlanetSim. Nevertheless, a more thorough study must be undertaken to study the feasibility of such integration. A C++ implementation of PlanetSim could also study the interoperability with NS [13] for example.

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 begin the simulation from that point. This feature is extremely useful for large simulations and saves valuable computing time.

Finally, the Network can be replaced by a Network Wrapper. This wrapper then assumes the tasks of the Simulator, and it routes incoming and outgoing

Node's messages using appropriate TCP or UDP connections on top of a real IP network. It is also responsible for calculating the proximity metric between nodes and to optimize the communication channels, disconnection events and specific timeouts of the underlying IP network. The NetworkWrapper thus allows moving unchanged simulated code to a real Internet testbed like PlanetLab. However, note that Network Wrapper provides different methods than Network, it does replaces completely the simulator in the interaction with nodes. NetworkWrapper does not include *simulate* a method nor inherits or implements any Network class. Also note than we are still working in the NetworkWrapper and much work remains to be done at this point.

3 Validation

In order to validate the PlanetSim framework we have implemented two structured overlays (Chord and Symphony) and several overlay services and applications (Scribe and DHT applications). We believe that our results confirm the generality, accuracy, and performance of our infrastructure.

Chord [11] is a classical structured ring-based topology that assures O(log n) lookup hops with pointers (finger table) to log (n) nodes where n is the number of nodes in the system. Chord's lookup mechanism is robust in the face of node failures and re-joins but it requires a periodic and costly stabilization protocol.

Our implementation of the Chord protocol aims to be close to MIT's Chord specifications and our results coincide with MIT published statistics. We however use a 32-bit address space in this paper for performance reasons –although the simulator can be configured to use a maximum of 160 bits–.

We also implemented the Symphony [12] overlay protocol in order to compare a deterministic approach to routing (Chord) to a probabilistic one (Symphony). Symphony is inspired by Kleinberg's Small World model and constructs a ring topology where each node has few long distance links. Symphony demonstrates that with $k = O(1)$ links per node, it is possible to route lookups with an average latency of $O(1/k \log^2 n)$.

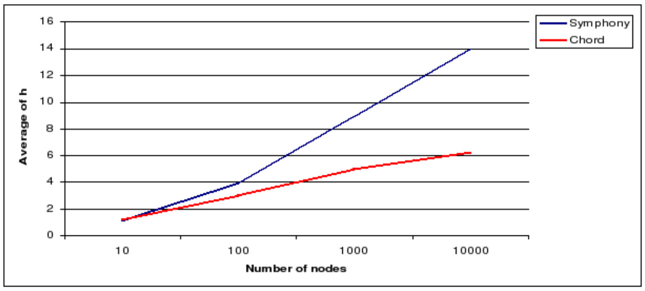


Fig. 3. Chord vs Symphony lookup hops

As we can see in figure 3, both algorithms scale gracefully with the increase in the number of nodes. Obviously, Chord performs better as a result of a bigger routing table and deterministic routing, but Symphony is less communication intensive with a very small maintenance algorithm. Like published results, Chord shows an average $1/2 \log_2(n)$ function and Symphony a $\log_{10}^2 (n)$ function.

Furthermore, we have implemented and tested an efficient overlay broadcast algorithm [3]. We obtained the awaited results where all nodes are covered in the broadcast process and that no redundant messages are sent.

As example application, we present here the Scribe [1] application level multicast protocol. Scribe is a large-scale and decentralized event notification system built on top of an overlay layer. The overlay layer, originally a Pastry network [10], is used to maintain topics and subscriptions, and to build efficient multicast trees. Scribe's randomized placement of topics and multicast roots balances the load among participating nodes.

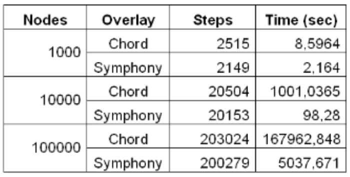
Simulation results indicate that Scribe scales well. It efficiently supports a large number of nodes, topics, and a wide range of subscribers per topic. Hence, Scribe can concurrently support applications with widely different characteristics. Results in our simulator also show that it balances the load among participating nodes, while achieving acceptable delay and link stress. Besides, implementing Scribe was straightforward by leveraging original FreePastry code based on the common API. Our layered approach also permits to test the Scribe algorithm in different overlays like Chord or Symphony. We do not show here graphical results due to lack of space.

3.1 Performance

One of the main goals of the PlanetSim framework is to achieve good performance for a high number of nodes. Due to the election of the Java language, we have been forced to spend a lot of resources in profiling and optimizing the simulator code. Besides, we have been faced with a constant compromise between clean designs and performing code.

Examples of such optimizations in our code are an efficient MessagePool that reuses messages, a custom Id class avoiding the Java's BigInteger, and static Singletons and Factories for loading Node, Message and Application types.

We run our experiments on 3 GHz (1 Gb RAM) Pentium 4 machines running Linux 2.4.24. We measured the time and steps required to stabilize Symphony



Nodes	Overlay	Steps	Time (sec)
1000	Chord	2515	8,5964
	Symphony	2149	2,164
10000	Chord	20504	1001,0365
	Symphony	20153	98,28
100000	Chord	203024	167962,848
	Symphony	200279	5037,671

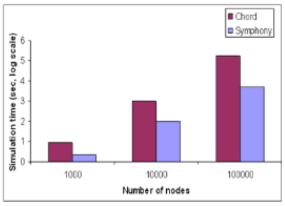


Fig. 4. Chord vs Symphony stabilization time

and Chord networks of different sizes. As we can see in Figure 4, Symphony performs much better than Chord in simulation time.

Chord needs around 8 seconds to stabilize a 1000 nodes network, 16 minutes for 1000 nodes and 46 hours for 100000 nodes. Symphony requires 2 seconds for 1000 nodes, 98 seconds for 10000 nodes, and 1.3 hours for 100000 nodes. Note that the bars are shown in a base 10 logarithmic scale to improve visualization. These results clearly show that the overhead imposed by the Chord stabilization protocol is quite big compared to Symphony's maintenance algorithms.

We believe that these numbers demonstrate the feasibility of using PlanetSim for large overlays. As future work, the distributed version of the simulator can even permit simulation of much higher number of nodes in an overlay network.

4 Related Work

First of all, we distinguish between network simulators and overlay simulators. The formers provide packet-level simulation of network protocols (TCP, UDP, IP, etc) over realistic Internet topologies. However, congestion-aware simulation including packet-loss and queuing delays is costly, leading to inappropriate scaling numbers for big overlays. Overlay simulators are usually more interested in evaluating overlay algorithms and its routing behaviour 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 example, the NS [13] network simulator provides a standard framework for accurate simulation of network protocols. NS is appropriate to simulate networks in the link, switching and transport layer but it is not aimed for application level overlays. Besides, for smaller scale scenarios NS performs gracefully, but for overlays over a hundred nodes in size suffers considerable scaling problems. Another example is the J-Sim [14] network simulation framework that follows a component oriented approach. Similar to ns-2, J-Sim is a dual-language simulation environment in which classes are written in Java (for ns-2, in C++) and "glued" together using Tcl/Java. Being easier to use than Ns-2, J-Sim also lacks enough scalability and performance for big overlays.

Other network simulators like SFFNET and OMNET++ have also been successfully used for peer to peer applications. Particularly, OMNET++ provides a rich environment that enables both packet-level simulations and high-level overlay protocols. Nevertheless, all these network simulators are mainly aimed for packet-level protocols, and impose additional complexity to the user learning curve.

In the end, many research groups have created their own overlay simulators, sacrificing accuracy for scale. Examples of these include p-sim, FreePasty, SimPastry, 3LS, PLP2P, and SimP^2. In the field of structured overlays, one of the pioneers is MIT's pspsim. This simulator currently supports many protocols, including Chord, Koorde, Kelips, Tapestry, and Kademlia. p2psim is protocol extensible, and it is 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.

FreePastry [10], the Java open-source implementation of the Pastry structured P2P protocol includes as well, the possibility to simulate applications on top of this overlay network. As in PlanetSim, FreePastry provides a Common API [2] to the applications built on top of it, 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. However, FreePastry is highly tied to the Pastry protocol, and it does not permit simulation of its applications on top of other structured P2P protocols.

Another interesting approach is the one followed by MACEDON [9]. 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 [13]. 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. 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.

5 Conclusions and Future Work

We have presented the PlanetSim overlay simulation/experimentation framework that facilitates design and implementation of both overlay algorithms and overlay distributed services. PlanetSim has been carefully designed to provide clean hotspots that make the framework extensible at all levels. Extensibility and external integration is a main goal of our framework because we believe that it is quite difficult to offer all the services that overlay researchers require.

Furthermore, our adoption of FreePastry's object oriented implementation of the Common API for structured overlays is a key aspect to ease the transition from simulation code to network code and vice versa. Unlike other simulators, we clearly distinguish between overlay algorithms (key based routing), and the applications and services built on top of them. Another side benefit of this design decision is that we can easily leverage FreePastry application code like Scribe and others.

We believe that PlanetSim can be used in peer to peer research settings but also as an educational tool to better understand overlay algorithms and services. Besides, the Network Wrapper code permit users to easily test their designs over the Internet using existing infrastructures like PlanetLab.

Of course, and like many other frameworks, PlanetSim can fail to attract users and developers in the research and educational settings. There is now a big inertia in the research arena towards custom-made simulators that solve particular problems. This is sad because it avoids clear comparisons in a unified platform. Besides, the framework cannot acquire critical mass without external contributors delivering new algorithms and services.

We however plan to extend the framework to incorporate new services and algorithms in the short term. We outline an improved overlay visualization engine for overlay networks and services, and a distributed version of the simulator enabling simulation of huge number of nodes (0,5M to 1M). PlanetSim is an open source project that is being actively used in our University for research and educational purposes. We welcome future collaborations or extensions to the project. PlanetSim is available with full source code and GPL license in http://ants.etse.urv.es/planet.

Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Technology through project TIC-2003-09288-C02-00.

References

1. Castro, M., Druschel, P., et al, "Scalable Application-level Anycast for Highly Dynamic Groups", *Proc. of NGC'03,* September 2003.
2. Dabek, F., Zhao, B.Y., Druschel, P., Kubiatowicz, J., and Stoica I., "Towards a Common API for Structured Peer-to-Peer Overlays", 2^{nd} *International Workshop on Peer-to-Peer Systems, IPTPS'03,* Berkeley, CA, February 2003.
3. El-Ansary, S.; Alima, L.O.; Brand, P.; et al. "Efficient Broadcast in Structured P2P Networks", 2^{nd} *International Workshop on Peer-to-Peer Systems, IPTPS'03,* Berkeley, CA, February 2003.
4. Gummadi, K., Saroiu, S., et al., "King: Estimating latency between arbitrary Internet end hosts", *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop.* Marseille, France, November 2002.
5. Medina, A., Lakhina, A., Matta, I., et al. "BRITE: An Approach to Universal Topology Generation", *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2001).* Cincinnati, Ohio, August 2001.

6. Pairot, C., García, P., Gómez Skarmeta, A.F., "DERMI: A Decentralized Peer-to-Peer Event-Based Object Middleware", *Proceedings of ICDCS'04*, Tokyo, Japan, pp. 236-243.
7. Pairot, C., García, P., Gómez Skarmeta, A.F., "Dermi: A New Distributed Hash Table-based Middleware Framework", *IEEE Internet Computing*, Vol. 8, No. 3, May/June 2004, pp. 74 – 84.
8. PeerSim Peer-to-Peer Simulator. http://peersim.sourceforge.net/
9. Rodriguez, A., Killian, C., Bhat, S., et al., "MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks", *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.
10. Rowstron, A., and Druschel, P., "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems", *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pp. 329-350, November 2001.
11. Stoica, I., Morris, D., Karger, D., et al. "Chord: A Scalable Peer-to-peer- Lookup Service for Internet Applications", *Proceedings of the ACM SIGCOMM 2001*, San Diego, CA, August 2001, pp. 149-160.
12. Singh, G.M., Bawa, M., Raghavan, P. "Symphony: Distributed Hashing in a Small World". *Proceedings of USITS'03*, Seattle, WA.
13. The Network Simulator – ns – 2. http://www.isi.edu/nsnam/ns/
14. J-Sim. http://www.j-sim.org/