An Overlay Network Simulation Framework
http://ants.etse.urv.es/planetsim

User and developer tutorial

Jordi Pujol Ahulló <jordi.pujol@estudiants.urv.es>
Marc Sánchez Artigas <marc.sanchez@urv.net>
Pedro García López <pedro.garcia@urv.net>

http://ants.etse.urv.es/planet

Contents

Figures

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.

The p2p systems and algorithms have evolved during the last years. From the early central index scheme used in Napster, and the flooding techniques of Gnutella, to the structured peer-to-peer key-based routing (KBR) overlays, there has been a huge leap. More specifically, these last ones (KBRs), are the kind of p2p networks which have been more active lately in terms of research. The particularity of KBR networks (also known as Distributed Hash Tables (DHTs), although this is only one of the abstractions KBRs usually provide) is that they follow a structured form (typically a ring or a tree), thus guaranteeing that any message routed from one node to another will not exceed the mean of O(log N) hops, where N is the total number of nodes in the network. This introduced determinism in p2p networks, since in unstructured networks (which usually use flooding techniques), it was very difficult to know whether a resource was available or not (recall that flooding is not a broadcast and as such, only a few leaves of the network are covered).

As we can see, we can thus 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 interconnected peers.

Current research in peer to peer systems is lacking appropriate environments for simulation and experimentation of large scale overlay services. 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, 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 are developing 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.

2 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 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 several thousands nodes in size suffers considerable scaling problems.

Another example is the J-Sim 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 p2psim. 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, 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 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. 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. 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.

3 Features

PlanetSim is a P2P overlay Simulator that simplifies development, testing and probing of distributed algorithms. As key advantages of our simulator we outline:

- It follows a layered and modular software architecture that facilitates framework extensions and it reduces the developer learning curve. PlanetSim has a clean architecture and it is based on well-known design patterns.

- It provides a Common API for application developers that help build distributed services on top of different overlay algorithms. This can represent a nice educational tool to explain and test overlay algorithms and services, and thus avoiding the complexities of implementing systems like Chord or Symphony.

- For overlay developers, we include a behaviours system that simplifies the implementation of message interchange protocols. This behaviours system also permits an advanced filtering mechanism that intercepts events related to specific patterns.

- PlanetSim also offers several output formats like Pajek or GML that can be used for overlay visualization and analysis. The output system is extensible and users can also generate their own graph formats.

- PlanetSim is not restricted to structured overlays. We will soon provide a simple Gnutella implementation on top of Planetsim.

4 Simulator 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.

4.1 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 ser-vices and applications on top of existing overlays. This API is based on the proposed Common API (CAPI) for structured Peer-to-Peer overlays published by Dabek et al. The main motivation for this decision is the plethora of applications and services that can be built on top of structured overlays.

In the paper, 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 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, content distribution networks and cooperative replication of archival storage. Furthermore, many traditional applications like Usenet or DNS have recently been rearchitected on top of these decentralized architectures.

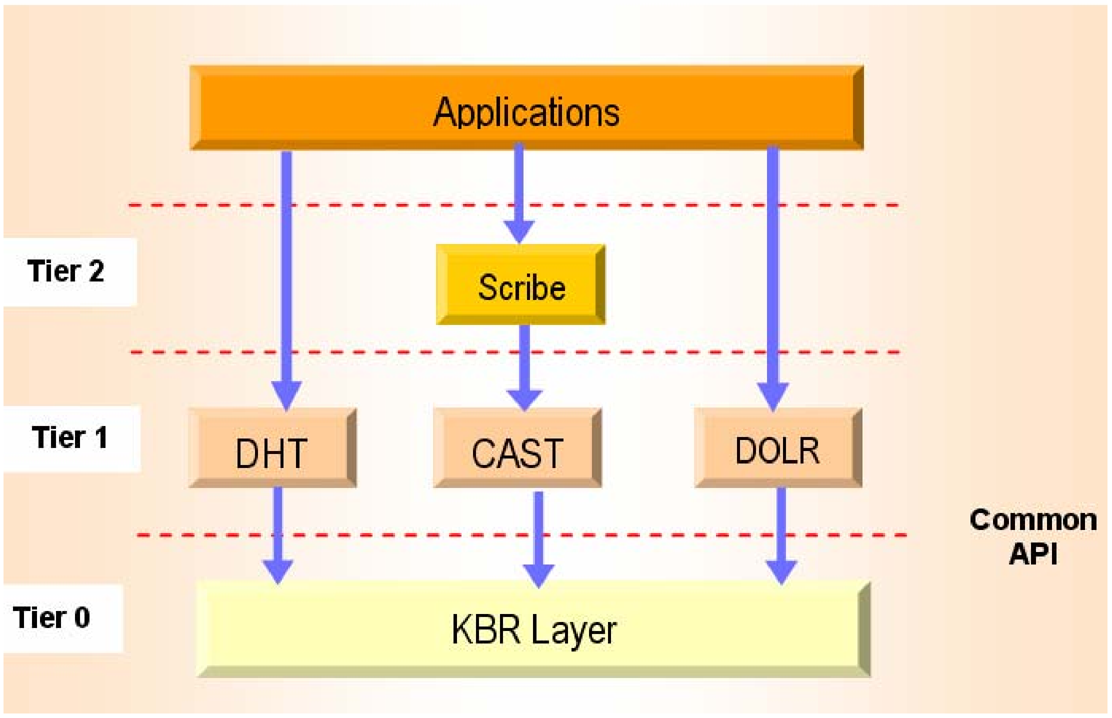

Figure 1. Common API Diagram

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 kinds 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 **localLookup**, **neighbourSet**, **replicaSet**, **update**, and **range**. They give information about routing state and identifier space information from running nodes.

Using these functions, the authors 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.

4.2 PlanetSim Layered Design

PlanetSim 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. At this moment, only the simulated Network is available.

We outline three main extension points (hotspots) in our framework:

- **Application**: Developers of overlay services like Scribe must implement the Application interface 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 implement the Node interface to incorporate the required overlay protocol. There is an abstract implementation name NodeImpl that 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 (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 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.

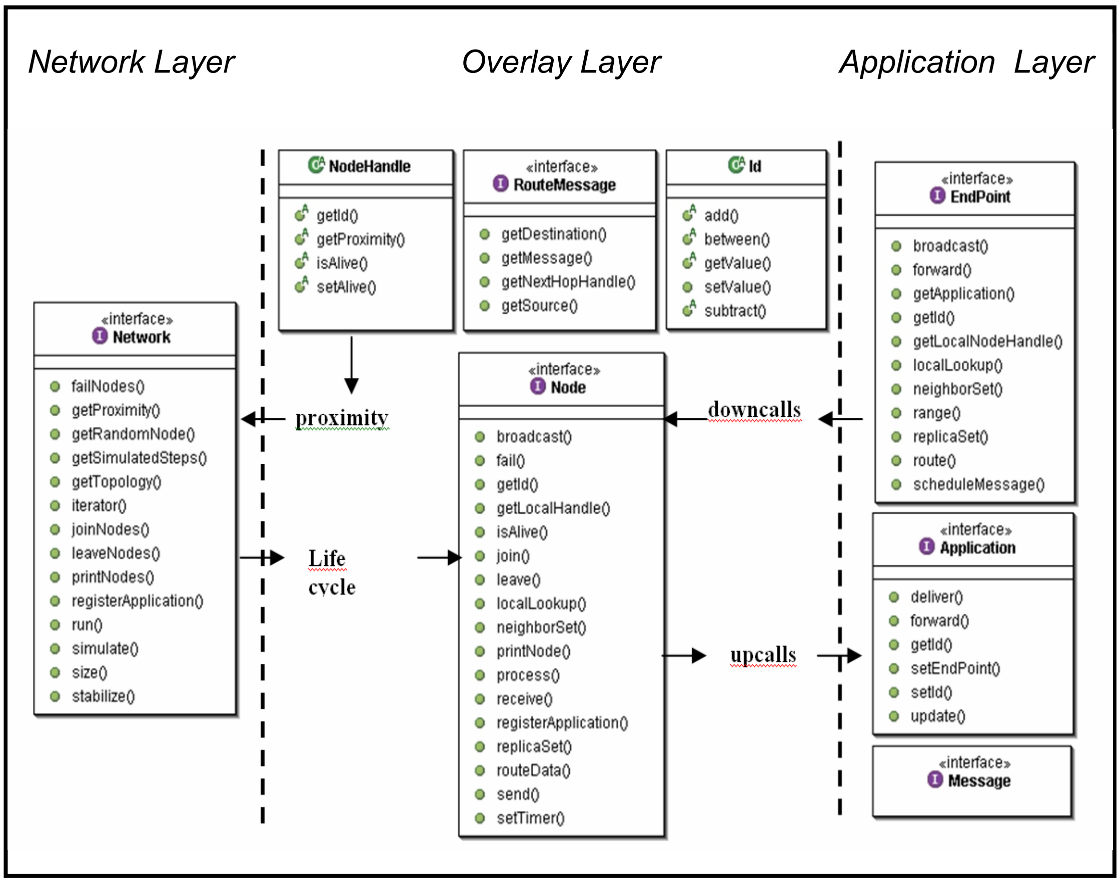

Figure 2. Layered Architecture

4.2.1 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. Read the Developer tutorial for more information.

4.2.2 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 (Chord uses integer numbers of 32 to 160 bits, for example) 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 theoretically 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. Note that proximity information is still not available in Planetsim 3.0.

4.2.3 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 can process events in different steps. Events are nodes joins, leaves or fails. 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 Inter-network 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, land-marks, 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 RandomNetwork or CircularNetwork that do not include latency costs. It is however feasible to incorporate Peersim or Brite 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 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 NetworkWrapper provides different methods than Network, it does replace completely the simulator in the interaction with nodes. NetworkWrapper does not include the simulate 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.

4.3 PlanetSim kernel

This is a graphical representation of the PlanetSim kernel:

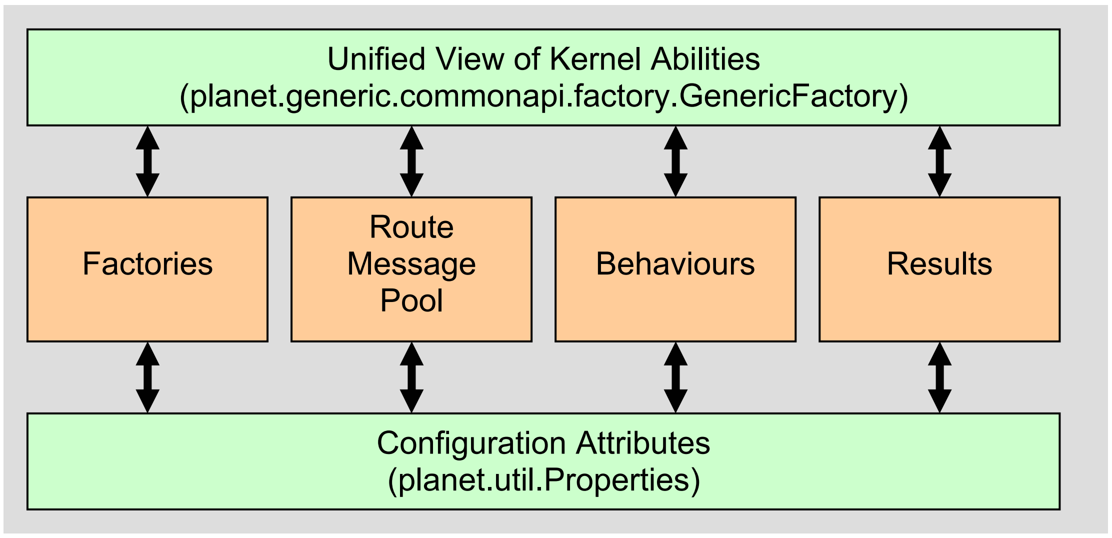

Figure 3. Simulator kernel

They have the following goals:

- **Configuration Attributes:** Its objective is to load the current specified configuration into the simulator context. The representative class is planet.util.Properties.

- **Factories:** It is a collection of classes that offer the ability of build generically instances of some type, including all items defined into the PlanetSim layers, as for example Ids, Nodes or NodeHandles. All these related methods follow the Factory Method design pattern.

- **RouteMessagePool:** In a whole network communication are required a lot of RouteMessages to make it stabilized. So, we have designed a pool to reuse them as possible. Nevertheless, on some critical point on the simulation will have a maximum of RouteMessage in use, where will be built the maximum number of RouteMessages. On the rest of the simulation all these messages will be reused.

- **Behaviours:** It defines a different operation schema for incoming RouteMessages process on the overlay implementation. The current distribution includes the Symphony overlay that uses this schema.

- **Results:** We have designed an alternative to make outputs using different formats, focused to show the graph formed with whole network. Currently are distributed the GML and Pajek results types.

- **Unified View of Kernel Abilities:** Because of the great number of instances to use simultaneously to accomplish a network simulation, we have designed a public layer with all required functionality to the developers. This view is based on the planet.generic.commonapi.factory.GenericFactory class, and occults all required instances and other little details.

4.3.1 Configuration Attributes

PlanetSim uses this schema to load the current configuration attributes:

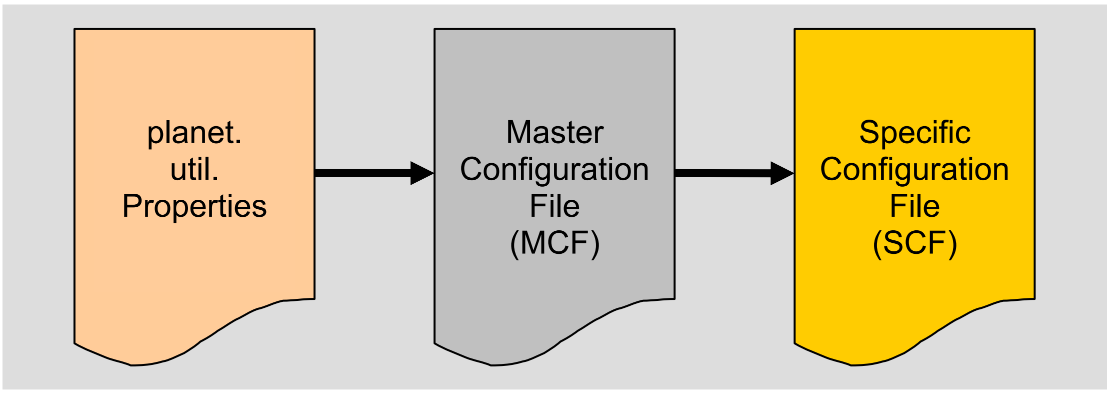

Figure 4. Loading Process of the Configuration

It uses two configuration files. The first MCF shows the SCF to use for each test. This file is only a bridge between a test and its configuration file. This schema has the advantage of to change the overlay to use in the current simulation, for example, without any source code modification and recompilation.

The SCF contains all detailed attributes for the PlanetSim kernel and other ones, related to the current test or the overlay in use.

4.3.2 Factories

Factories are a collection of interfaces and their implementation that permits instances building, using their methods that follow the Factory Method design pattern.

As for example, the IdFactory defines different methods for building new node Ids with certain values (primitives or objects). All these methods returns Id instances but don't any specific implementation (as ChordId or SymphonyId). With this ability, PlanetSim can build networks with different overlays and different implementations.

There is a Factory definition for each element that appears in the PlanetSim layers, as you can see in the planet.commonapi.factory package. They are the following:

- NetworkFactory
- NodeFactory
- NodeHandleFactory
- IdFactory
- EndPointFactory
- ApplicationFactory

Their names just define the instance type returned. But, to make it possible just need the configuration attributes explained above.

4.3.3 RouteMessagePool

It is a Factory and a Pool, all in one. This functionality is required because of the intensive use of RouteMessages on any simulation. If no pooling was made, the overhead for the Garbage Collector was increased visibly.

For the correct operation of this RouteMessagePool, the developer must ensure that all unused RouteMessages are free into the pool, and always the RouteMessages are getting from it.

4.3.4 Behaviours

In order to provide a greatest degree of reusability, PlanetSim provides a mechanism to organize the actions taken at node level. This mechanism is based on the notion of behaviour. Strictly speaking, behaviour is a class that let nodes perform an action in response of an incoming message. For the developer's viewpoint, behaviour is a piece of code that encapsulates an action that must be performed by the node when a suitable message arrives. By a suitable message we mean a message whose performative, that is, the type and the mode of the message matches the behaviour descriptor. The behaviour descriptor is an expression used to specify when a behaviour must be executed. In its most primary form, a behaviour descriptor can be a pair of literals corresponding to the type and a mode of a message. Nonetheless, behaviour descriptors can be more complex and accept several wildcards as we will see later in this chapter.

The core idea behind the use of behaviours is let the PlanetSim programmer hand-code different actions and use them as interchangeable pieces like a Lego artefact. For example, by binding a particular behaviour with a message, and later, swapping it by a new one without modifying the node's source code all the time. This let, via a configuration file, add and remove behaviours, that is, what a node must do without recompiling it again. Briefly, this configuration file has a new line for every behaviour entry. Every behaviour entry specifies the java class that encapsulates the behaviour and its descriptor that specifies when the behaviour must be executed.

4.3.4.1 Runtime Execution

Until now, we have introduced the notion of behaviour and its advantages but we don't have explained what happens at runtime when the simulator uses behaviours to model the p2p nodes involved in it.

Basically, a singleton object called behaviour's pool is loaded into the simulator. The behaviour's pool has the instances of the behaviours (an instance of each one for the whole simulator) and acts as a proxy executing the corresponding ones on the nodes that have new messages. In fact, the implementation of the behaviour's pool is not fixed and a programmer can customize a new one for its own interests. For that purpose, the simulator includes several interfaces (BehavioursFactory, BehavioursPool,…) to let developers customize the runtime behaviour classes. Nevertheless, the way to do it is out of the scope of this tutorial.

Upon the behaviour's pool falls the core task of the behaviour's infrastructure, we will explain on follows a little example to see the whole process. By now, we consider that a structured p2p overlay like Chord wants to replicate the contents stored under a key when a REPLICATE message arrives. Until now, PlanetSim users would probably make an implementation of such operation by modifying the **dispatcher** method inside the node. However, with this new approach in mind, a programmer would probably implement this new task in a new behaviour, called for instance, ReplicateBehaviour. Furthermore, we imagine that the latter one is what the programmer decided to do. In that case, once the programmer had finished the implementation of the ReplicateBehaviour, we would edit the configuration file and would include a new behaviour entry specifying that when a REPLICATE message arrives the ReplicateBehaviour must be executed. Thereafter, the programmer would run a new simulation.

The simulation proceeds as follows. At the start up, the simulator instantiates the behaviour's pool and loads the behaviours included in the configuration file. Then, the behaviour's pool is ready to begin invoking behaviours. This is occurs by intercepting the incoming messages and checking them against the behaviour descriptors. An interesting feature of the current behaviour's pool implementation is that a single message can match more than one behaviour descriptor. For that reason, the behaviour's pool keeps a stack of behaviours for every possible message at node level. These stacks have the behaviour instances ordered from more specific to more generic. In fact, if the protocol tends to perform common tasks for every new message arrival multiple behaviour invocations will incur frequently.

The behaviour's pool invokes a behaviour by passing a couple of arguments. These are the original RouteMessage and the reference to the node to who the RouteMessage was addressed. This reference allows updating the node's internal state to reflect the last network transaction. Once the behaviour execution finishes, the behaviour's pool returns the control to the node or either spawns a new behaviour depending on whether the stack of behaviours is over or not.

In the context of our example, when the behaviour's pool intercepts a REPLICATE message will dispatch the ReplicateBehaviour and finally, it will yield the control to the node.

4.3.5 Results

Into PlanetSim has been added new functionality to get outputs focused to represent the network topology as a graph. Examples are GML and Pajek formats.

This task involves to the Node implementation to get its connectivity as edges into a graph and a ResultsGenerator that writes the output into an external file, with the required format for that results type.

Different results type can coexist and use them into the same simulation to extract the graph information with different formats. Also, this operation schema is extensible and can add new formats implementing only new ResultsGenerator in the most of cases.

These are two examples of graph representation with different output formats:

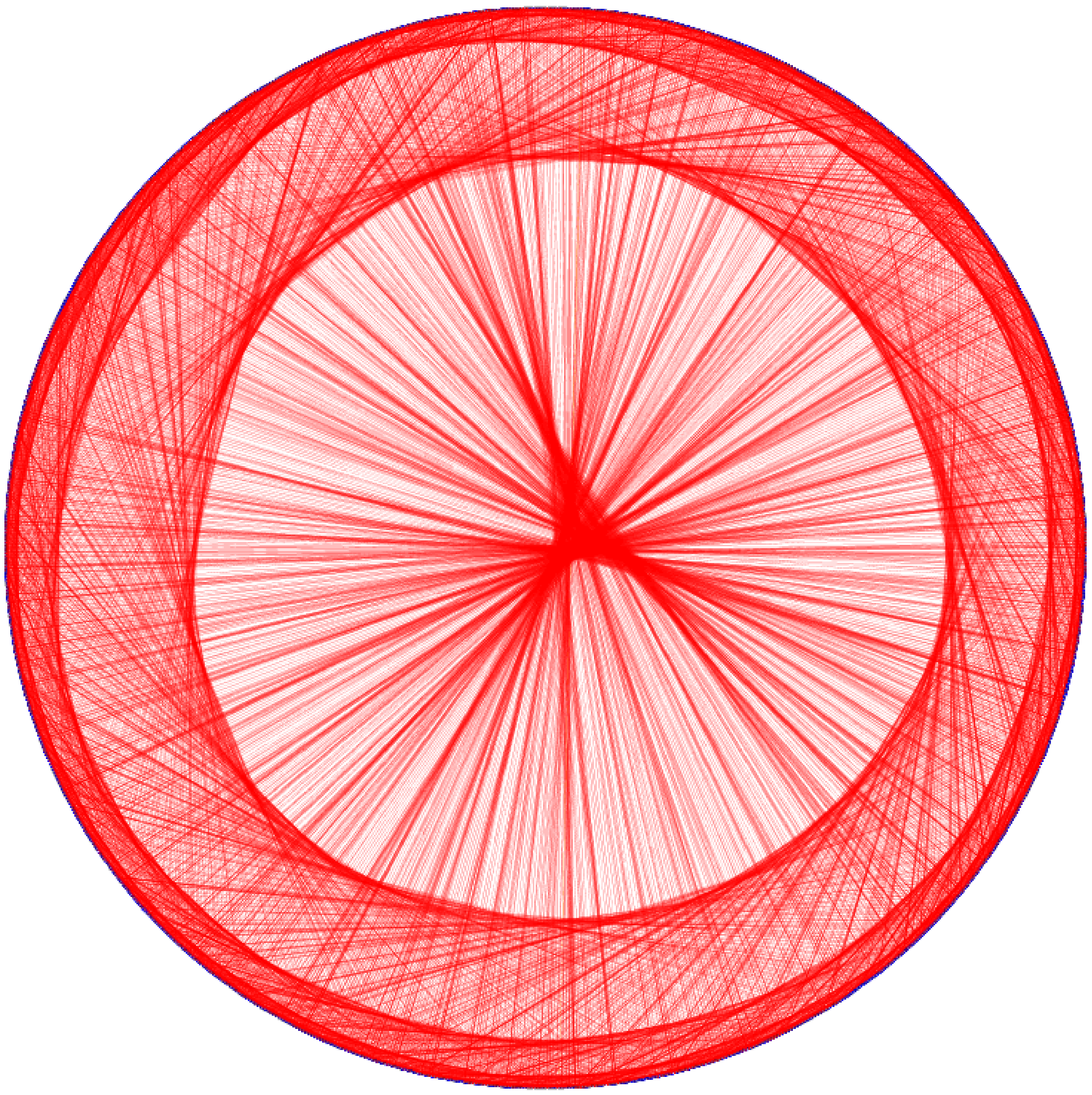

Figure 5. A Chord network with 1000 nodes, which their Ids are randomly built

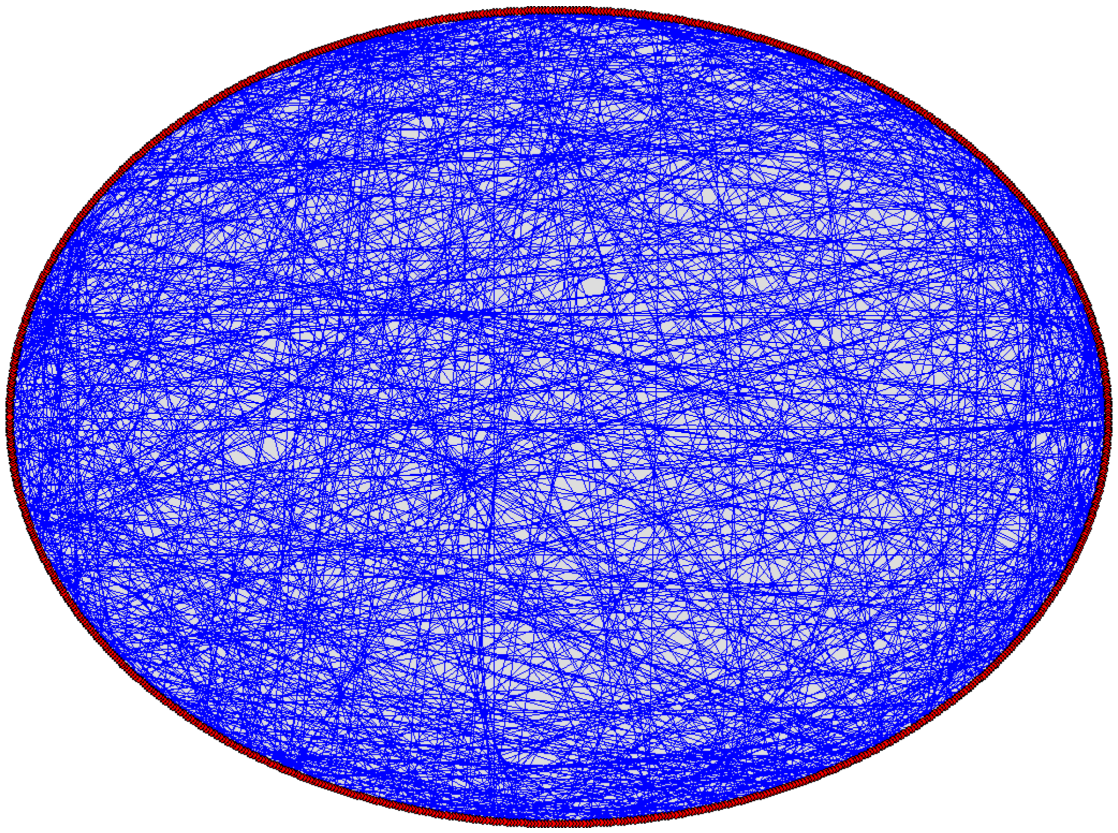

Figure 6. A Symphony network with 1000 nodes, which their Ids are randomly built

5 Developer Tutorial

With the following lines you will be able to follow how work as a developer within PlanetSim.

5.1 Main application: Basic structure

You need to understand the basic design of the main application and the configuration files used to run a complete simulation test. See the following figure:

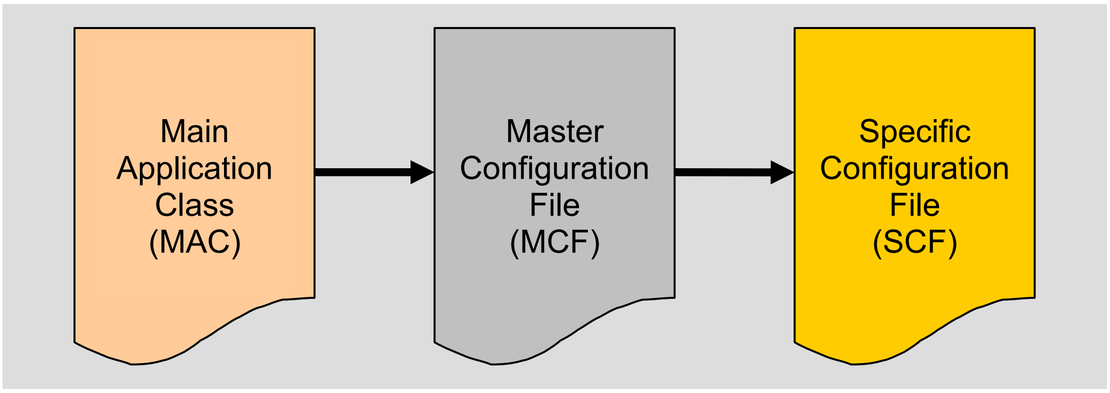
Main Application Class (MAC)
Master Configuration File (MCF)
Specific Configuration File (SCF)

Figure 7. Basic structure for a main application

MAC is the class that contains the **main()** method and executes the required test. **MCF** is a bridge between each main application and the desired configuration file. This contains a list of (key, value) pairs, where the key is the **unique name of the test** and the value is the path to **SCF**. The **SCF** contains all required attributes that define all values (classes, numbers, names,…) to use within the current test.

5.1.1 Building the Main Application Class

There are two ways to mount a main application and both using the **planet.generic.commonapi.GenericApp** class:

- Building a class that extends of GenericApp.
- Building a class and using the static methods of GenericApp.

This is an example of the first way, extending of GenericApp:

```java
package planet.test;

import planet.generic.commonapi.GenericApp;

public class Test extends GenericApp {

    public Test() throws Exception {
        super("../conf/master.properties", "TEST",
            false, false, false, false);

        //here your own code
    }

    public static void main(String[] args) throws Exception {
        new Test();
    }
}
```

Figure 8. Main application extending of GenericApp

This is an example of the second way, using the GenericApp class directly:

```java
package planet.test;

import planet.generic.commonapi.GenericApp;

public class Test2 {

    public static void main(String[] args) throws Exception {
        GenericApp.start("../conf/master.properties", "TEST2",
            false, false, false, false);

        //here your own code
    }
}
```

Figure 9. Main application using GenericApp

The default constructor or the **GenericApp.start()** method needs the following parameters:

1. The path to the MCF.
2. A unique name that identifies the current test. This will appear into the MCF as a key.
3. Flag to activate Application level properties.
4. Flag to activate events properties.
5. Flag to activate results properties.
6. Flag to activate serialization abilities.

The values of these last four parameters are test dependant, and identify when to load optional parts of the current configuration.

Following this description, into the MCF **PLANETSIM/conf/master.properties** will appear lines as follows:

```
TEST = ../conf/chord.properties
TEST2 = ../conf/chord.properties
```

Figure 10. Example of lines into MCF

These lines are only an example. They show the configuration file with all current attributes values to use within both tests. The user of these tests must ensure that the attributes appeared within the SCF contains the required configuration values to execute it correctly. The file specification may be as an absolute or relative path.

5.1.2 Specifying new configuration within MCF

There are situations where loaded default properties from the configuration file must be overwritten. On these cases you have to reconfigure the simulator context. The way is as follows:

```
package planet.test;

import planet.generic.commonapi.GenericApp;
import planet.generic.commonapi.factory.Topology;
import planet.util.Properties;

public class Test {

    public static void main(String[] args) throws Exception {
        GenericApp.start("../conf/master.properties","TEST",
            false,false,false,false);

        //example of overwritten attributes
        Properties.factoriesNetworkSize = 100;
        Properties.factoriesNetworkTopology = Topology.RANDOM;

        //reconfiguration of the simulator
        GenericApp.restart(false,false,false,false);

        //here your own code
    }
}
```

Figure 11. Reconfiguration of the simulator context

GenericApp.restart() method reloads all simulator context attributes to apply new values explicitly set. The four parameters are the same as the last four in the **GenericApp.start()** method or default constructor seen above.

After these lines of initialization of the simulator context appears the desired test properly.

5.2 Layer 2: The Application Layer

To build an Application within the simulator you have to do the following:

1. Build a new class that implements the **planet.commonapi.Application** interface.
2. Into this new Application you have to implement its functionality, as a DHT for example.
3. Build a new class that implements the **planet.commonapi.Message** interface, which will contain any required data to send between Applications in their normal operation.

5.2.1 Application example

The following figure is an example of an Application implementation:

```java
package planet.test.helloworld;
import planet.commonapi.*;
import planet.commonapi.exception.InitializationException;
import planet.generic.commonapi.factory.GenericFactory;
import java.util.*;

public class DHTApplication implements Application {
	private EndPoint endPoint = null;
	/**
	 * Identification of the application.
	 */
	public static String applicationId = "DHTApplication";
	/**
	 * Identification of the application instance.
	 */
	private String appId = applicationId;

	/**
	 * Constructor
	 */
	public DHTApplication() {
	}

	public void byStep(){}

	public void setEndPoint(EndPoint ep) {
		endPoint = ep;
	}

	public boolean forward(Message message) {
		System.out.println("[" + appId + "] over [" +
		    endPoint.getId()+ "]: Forwarding message...");
		return true;
	}
```

```
      public void deliver(Id id, Message message) {
            if (message instanceof DHTPeerTestMessage) {
                  DHTPeerTestMessage mesg =
                      (DHTPeerTestMessage) message;
                  System.out.println("Delivered Message: "
                              + ((DHTPeerTestMessage)
                                message).getData());
                  System.out.println("Destination Node : " +
                              this.endPoint.getId());
                  System.out.println("Message Id        : " + id);
            }
      }

      public String getId() {
            return appId;
      }

      public void setId(String appId) {
            this.appId = appId;
      }

      public void update(NodeHandle node, boolean joined) {
      }

      public Application setValues(String applicationName)
      {
          this.appId = applicationName;
          return this;
      }

      //application dependant methods
      public Message makeTestMessage(String data) {
            return new DHTPeerTestMessage(data);
      }

      public void send(String textKey, DHTPeerTestMessage mess) {
            try {
                  endPoint.route(GenericFactory.buildKey(textKey),
                      mess, null);
            } catch (InitializationException e) {
                  System.out.println(
                        "Cannot to be sent the message [" + mess
                        + "] with this key [" + textKey + "]");
                  e.printStackTrace();
            }
      }
}
```

Figure 12. HelloWorld Application example

There are the followings requirements within the implementation (marked as bold above):

1. Contain an **EndPoint** reference. The method **setEndPoint()** is invoked automatically when an Application instance is registered to a Node.
2. Contain a **String** attribute to save the **Application name**. One Node can contain very Applications at the same time, and this name is used to identify them. With the

default constructor this name has to be initialized (or in its declaration).

3. The **setValues()** method only will be invoked by the own simulator if the Application name has to be replaced.
4. The **forward()** method always have to return true, by default. This method is invoked when an Application level Message is routed into the underlying overlay, on each Node, including the destination Node.
5. The **deliver()** method is invoked when the Message has arrived to the destination Node. Its implementation has to process the incoming Message, making the required actions.

Other methods:

6. The **byStep()** method is invoked always by the underlying Node, when it just has finished its internal operations.
7. The **update()** method informs to this Application when a Node has joined (when true) or leaved (when false) to the network.

Other available functionality can be found by the EndPoint attribute, which offers routing and current connectivity information basically.

5.2.2 Message example

Following the same test used above (that appears into the current distribution), this is the Message implementation class for the HelloWorld test:

```java
package planet.test.helloworld;
import planet.commonapi.*;

public class DHTPeerTestMessage implements Message {
      /**
       * Contents of this message.
       */
      private String data = null;

      public DHTPeerTestMessage(String data) {
            this.data = data;
      }

      public String getData() {
            return data;
      }
}
```

Figure 13. Message implementation example

The Message interface contains no method. For this reason, all the methods and constructors that appear in this example is Application dependant.

5.2.3 Configuration attributes to modify

Once the Application and Message has been correctly implemented, there is another step to use these classes into a test.

You have to open the SCF for the current test and modify the entry `FACTORIES_APPLICATION` writing the fully qualified class name of your Application implementation. No Message entry is required because its management is made internally into the own Application implementation.

5.3 Layer 1: The Overlay Layer

How can I add a new overlay into this simulator? How can I fix its internal operation and routing management? Well, you should follow these steps:

1. Build a new class that implements the **planet.commonapi.Node** interface. Because there are some common aspects between overlays, we have released the **planet.generic.commonapi.NodeImpl** abstract class to extend by any new overlay implementation. However, if its functionality is different to the required, you can implement directly the **Node** interface.
2. Decide if its implementation is behaviours or programmationally based.
3. If your implementation is behaviours based, you have to build some behaviours to implement the internal overlay management (successors maintenance, topology maintenance, connectivity management, …), implementing the **planet.commonapi.behaviours. Behaviour**.
4. If your implementation is programmationally based, you have to include whole overlay management into the Node implementation.
5. Because of communication between Nodes is made with RouteMessages, you should to specify different types and modes for these communications.
6. Build a new class that extends of **planet.commonapi.Id**, an abstract class that defines the unique identification for any Node into the simulator.

7. Any important attribute of your overlay, common for all nodes, should be parametrical from the external SCF (see the Chord, Symphony or Trivial P2P parts in SCFs on the PLANETSIM/conf/ directory to take examples). These attributes have to be included into specific implementation of the **planet.util.OverlayProperties** for the current overlay.

5.3.1 Node example (Trivial P2P overlay)

To show a basic example of overlay, we have built the Trivial P2P. This is a ring based topology network, with only a successor and predecessor as links per Node. Its implementation offers the ability to work with or without behaviours.

See the TrivialNode implementation:

```
package planet.trivialp2p;

import java.util.Collection;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Set;
import java.util.Vector;

import planet.commonapi.Id;
import planet.commonapi.Message;
import planet.commonapi.NodeHandle;
import planet.commonapi.RouteMessage;
import planet.commonapi.behaviours.BehavioursPool;
import planet.commonapi.exception.InitializationException;
import planet.commonapi.results.ResultsConstraint;
import planet.commonapi.results.ResultsEdge;
import planet.generic.commonapi.NodeImpl;
import planet.generic.commonapi.factory.GenericFactory;
import planet.simulate.Results;
import planet.util.Properties;

public class TrivialNode extends NodeImpl {

    /* *************  CONSTANTS FOR MODE OF ROUTEMESSAGE *******/
    public final static int REQUEST                       = 0;
    public final static int REFRESH                       = 1;

    /* END ******  CONSTANTS FOR MODE OF ROUTEMESSAGE *******/

    /* **********  CONSTANTS FOR TYPE OF ROUTEMESSAGE *******/
    public final static int DATA                            = 0;
    /* END ******  CONSTANTS FOR TYPE OF ROUTEMESSAGE *******/

    /* **********  CONSTANTS FOR TYPE/MODE OF ROUTEMESSAGE *******/
    public final static String[] TYPES = { "DATA" };
    public final static String[] MODES = { "REQUEST", "REFRESH" };
    /* END ******  CONSTANTS FOR TYPE/MODE OF ROUTEMESSAGE *******/

    // Routing table:
    /** The successor of the actual node. */
    private NodeHandle successor;
    /** The predecessor of the actual node. */
    private NodeHandle predecessor;
    /** Contains ALL links of the actual node. */
    private Set links;
    /** Contains the unique node successor. */
    private Vector successors;
    /** If true, the node is already alive. */
    private boolean alive;
    /** The behaviours pool to be used. */
    private BehavioursPool behPool;


    /* ******* STARTING IMPLEMENTATION *************************/

    /**
     * Initialize the internal structure.
     */
    public TrivialNode() throws InitializationException {
        super();
        alive = true;
        successor = null;
        predecessor = null;
        links = new HashSet(2);
        successors = new Vector(1);
        if (Properties.overlayWithBehaviours)
            behPool = GenericFactory.getDefaultBehavioursPool();
    }

    /**
     * Nothing does. This implementation don't contain a
     * stabilization protocol.
     * @param bootstrap Bootstrap node.
     * @see planet.commonapi.Node#join(planet.commonapi.NodeHandle)
     */
    public void join(NodeHandle bootstrap) {
    }

    /**
     * Nothing does. Only sets the alive flag to false.
     * @see planet.commonapi.Node#leave()
     */
    public void leave() {
        alive = false;
    }

    /**
     * Gets the internal routing information in a hashtable.
     * The key informs the concept of the related value.
     * @return A hashtable with the internal routing information.
     * @see planet.generic.commonapi.NodeImpl#getInfo()
     */
    public Hashtable getInfo() {
        Hashtable info = new Hashtable();
        info.put("successor",successor);
        info.put("predecessor",predecessor);
        return info;
    }

    /**
     * Returns the own nodehandle or its successor nodehandle, in a
     * clockwise proximity.
     * @param id The id to be find.
     * @return The nearest nodehandle in a clockwise manner.
     * @see
planet.commonapi.Node#getClosestNodeHandle(planet.commonapi.Id)
     */
    public NodeHandle getClosestNodeHandle(Id id) {
        return (predecessor.getId().
            betweenE(predecessor.getId(),this.id)) ?
                    this.nodeHandle :
                    successor;
    }

    /**
     * Routes an application level message to the destination node.
     * @param appId Application name.
     * @param to Destination node (or key).
     * @param nextHop May be null. The next hop into the route.
     * @param msg Application level message to be sent.
     * @see planet.commonapi.Node#routeData(java.lang.String,
planet.commonapi.NodeHandle, planet.commonapi.NodeHandle,
planet.commonapi.Message)
     */
    public void routeData(String appId, NodeHandle to, NodeHandle
nextHop, Message msg) {
        RouteMessage data =
            buildMessage(GenericFactory.generateKey(),
            nodeHandle,to,nextHop,DATA,REQUEST,appId,msg);
        if (data!=null)
        {
            Results.incTraffic();
            this.dispatchDataMessage(data,REQUEST,REFRESH);
        }
    }

    /**
     * Do nothing. Only sets to false the alive flag.
     * @see planet.commonapi.Node#fail()
     */
    public void fail() {
        alive = false;
    }

    /**
     * Prints out the routing information of this node.
     * @see planet.commonapi.Node#printNode()
     */
    public void printNode() {
        System.out.println("<Node id=\""+id+"\">");
        System.out.println("   <Successor
id=\""+successor.getId()+"\">");
        System.out.println("   <Predecessor
id=\""+predecessor.getId()+"\">");
        System.out.println("</Node>");
    }
    /**
     * Prints out the local node information.
     * @see planet.commonapi.Node#prettyPrintNode()
     */
    public void prettyPrintNode() {
        System.out.println("<Node id=\""+id+"\"/>");
    }

    public void broadcast(String appId, NodeHandle to, NodeHandle
            nextHop, Message msg) {
        throw new NoSuchMethodError("Method not implemented yet.");
    }

    public NodeHandle getPred() {
        return predecessor;
    }

    public NodeHandle getSucc() {
        return successor;
    }

    public boolean isAlive() {
        return alive;
    }

    public Vector getSuccList(int max) {
        //NOTE: only exists one successor
        return successors;
    }

    public Vector localLookup(Id key, int max, boolean safe) {
        return null;
    }

    public Vector neighborSet(int max) {
        return null;
    }

    public Vector replicaSet(Id key, int maxRank) {
        return null;
    }

    public boolean range(NodeHandle node, Id rank, Id leftKey, Id
rightKey) {
        return false;
    }

    /**
     * Build the edges for its sucessor and predecessor links.
     * @param resultName Result name to be used.
     * @param edgeCollection Edge collection where to add all the new
ones.
     * @param constraint Constraint to verify the addition of the
edges.
     * @see planet.commonapi.Node#buildEdges(java.lang.String,
java.util.Collection, planet.commonapi.results.ResultsConstraint)
     */
    public void buildEdges(String resultName, Collection
edgeCollection, ResultsConstraint constraint) {
        if (edgeCollection == null || constraint == null) return;
        //neighbours (successors and predecessors)
        ResultsEdge e =
buildNewEdge(resultName,id,successor.getId(),"#0000FF");
        if (e!=null)
            if (constraint.isACompliantEdge(e))
edgeCollection.add(e);
        e =
buildNewEdge(resultName,id,predecessor.getId(),"#0000FF");
        if (e!=null)
            if (constraint.isACompliantEdge(e))
edgeCollection.add(e);
    }

    public Set getAllLinks() {
        return this.links;
    }

    /**
     * Process the local incoming messages.
     * @param actualStep Actual step in the simulation process.
     * @return Always false, whenever the node always is stabilized
     * and don't require more steps for its stabilization.
     * @see planet.commonapi.Node#process(int)
     */
    public boolean process(int actualStep) {
        //always must be invoked at the beginning
        super.process(actualStep);

        //here starts your node process
        if (Properties.overlayWithBehaviours)
        {
            //you may use this structure when your implemented
            //overlay use behaviours
            dispatchMessagesWithBehaviours();
        } else {
            //you may use this structure when your implemented
            //overlay don't use behaviours
            dispatchMessages();
        }

        //always must be invoked at the end
        invokeByStepToAllApplications();
        return false;
    }

    public Node setValues(Id newId) throws InitializationException {
        super.setValues(newId);
        //this overlay doesn't require any other action.
        return this;
    }

    /* ******** SPECIFIC OVERLAY METHODS ************************/

    /**
     * Updates the node predecessor.
     * @param pred The new node predecessor.
     */
    public void setPredecessor(NodeHandle pred)
    {
        if (predecessor!=null)
            links.remove(predecessor);
        predecessor = pred;
        links.add(pred);
    }

    /**
     * Updates the node successor.
     * @param succ The new node successor.
     */
    public void setSuccessor(NodeHandle succ)
    {
        if (successor!=null)
        {
            links.remove(successor);
            successors.remove(0);
        }
        successor = succ;
        links.add(succ);
        successors.add(succ);
    }

    /**
     * Dispatch all incoming messages of applicaion level.
     */
    private void dispatchMessages()
    {
        while (hasMoreMessages())
        {
            RouteMessage msg = nextMessage();
            //Only application level messages are delivered
            dispatchDataMessage(msg,REQUEST,REFRESH);
        }
    }

    /**
     * Dispatch all incoming messages of application level using
     * behaviours.
     *
     */
    private void dispatchMessagesWithBehaviours()
    {
        while (hasMoreMessages()) {
            RouteMessage msg = nextMessage();
            try {
                behPool.onMessage(msg, this);
                GenericFactory.freeMessage(msg);
            } catch
(planet.commonapi.behaviours.exception.NoSuchBehaviourException e) {
                throw new
                   Error("An applicable behaviour is not found");
            } catch
(planet.commonapi.behaviours.exception.NoBehaviourDispatchedException
d) {
                throw new
                   Error("An applicable behaviour is not found");
            }
        }
    }

    /* END **** SPECIFIC OVERLAY METHODS ************************/

    public String toString()
    {
        return "<TrivialNode id=\""+id+"\">";
    }
}
```

Figure 14. Node implementation for the Trivial P2P overlay

There are important points in this basic implementation:

1. All the types and modes for the communications on current overlay appear as a **final static** attributes into the specific Node implementation. In this case only for the Application level communication.
2. Because of this TrivialNode extends of **NodeImpl**, appears the **super()** statement in the default constructor.
3. Into the **setValues()** method only invokes the default implementation of the NodeImpl. But if it requires, your own initialization should put into this method.
4. The initialization of any Node is made by this sequence of actions: 1) invoking the **default constructor**, and 2) invoking the **setValues()** method.
5. The **process()** method shows exactly the header and footer of this method, and also shows the possible ways to implements the incoming RouteMessages treatment.
6. The **dispatchMessages()** method shows the way to implement procedurally the RouteMessages dispatching.
7. The **dispatchMessagesWithBehaviours()** method shows the way to implement the RouteMessages dispatching using behaviours.

This implementation of RouteMessages dispatching would not confuse you. In this case there are only RouteMessages for the Application level Messages, because this overlay is **directly** stabilized (see the **planet.test.trivialp2ptest** for more details). But it is applicable to any other (and more common) situation.

For the Id into the Trivial P2P overlay is used the existing SymphonyId (a double value based Id). You can do this. Reuse any other good implementation to accelerate your job.

The OverlayProperties for this Trivial P2P is as follows:

```
package planet.trivialp2p;

import planet.commonapi.exception.InitializationException;
import planet.util.OverlayProperties;
import planet.util.PropertiesWrapper;

public class TrivialProperties implements OverlayProperties {

      /* *********** TRIVIALP2P PROPERTIES ***********************/
      /* Theese must to appear on the properties file */
```

```
	/**
	 * TrivialP2P property: Default key for 'debug' flag.
	 */
	public static final String TRIVIAL_DEBUG = "TRIVIAL_DEBUG";
	/* ********* TRIVIALP2P ATTRIBUTES   *********************/
	/**
	 * When true, shows information for debug purposes.
	 */
	public boolean debug;


	public void init(PropertiesWrapper properties) throws
InitializationException {
		//Load properties
		debug = properties.getPropertyAsBoolean(TRIVIAL_DEBUG);
	}

	public void postinit(PropertiesWrapper properties) throws
InitializationException
	{
		//does nothing
	}

	/**
	 * Returns a String representation of the constant specific
	 * values of type the RouteMessage. Its use is only for human
	 * readable logs. Based on SymphonyNode implementation.
	 * @param type Value to get its String representation.
	 * @return The String representation of the type.
	 */
	public String typeToString(int type) {
		return TrivialNode.TYPES[type];
	}

	/**
	 * Returns a string representation of each of event mode and
	 * RouteMessage mode.
	 * @param mode Mode of the RouteMessage to get its
	 * String representation.
	 * @return String representation of the mode of RouteMessage.
	 */
	public String modeToString(int mode) {
		return TrivialNode.MODES[mode];
	}

	/**
	 * Returns RouteMessage type for Application level.
	 * @return RouteMessage type for Application level.
	 */
	public int getTypeForApplicationMessage()
	{
	    return TrivialNode.DATA;
	}
}
```

Figure 15. OverlayProperties implementation for Trivial P2P Overlay

Its initialization is made in two steps. First is invoked the **init()** method and loads all available properties without using (if it is required) the **GenericFactory.buildXXX()** methods. At the second step the **postinit()** method is invoked, when the GenericFactory is correctly initialized and available all its methods.

The **typeToString()** offers the ability to show a String representation of any existing RouteMessage type. The **modeToString()** does the same, but for the RouteMessage modes.

The **getTypeForApplicationMessage()** returns the type for the Application level communications.

6 Getting Results

PlanetSim offers the ability for getting results. There are two ways:

1. Showing the Node information for whole network. It is based on **Node.prettyPrintNode()** and **Node.printNode()** methods. Its style is free and as text.
2. Showing the current state of the network, with the nodes connectivity. In the current distribution we include the GML and Pajek outputs. Its style follows the required formats to load graphs into the related viewers.

For the first way, we have included an automatic management of this output. In the SCF related to the current test, an entry exists with the name `SIMULATOR_PRINT_LEVEL`. This has three values: 0 for no output, 1 for Node.prettyPrintNode() invocation and 2 for full printing with the Node.printNode() invocation. To get this output type at any moment, **always to the default output (System.out)**, you should put this line into your code:

```
GenericApp.printNetwork(net);
```

where the **net** parameter is the current Network instance.

The way to get a GML or Pajek output is "quite more complex". See the following example:

```
GenericFactory.generateResults(ResultsNames.GML,network,
    "network.gml", GenericFactory.buildConstraint(
    ResultsNames.GML), true);
```

This statement produces the GML output into the **network.gml** file. This is the parameter description:

1. Result type name.
2. The current Network instance.
3. Filename where to write the output.
4. The desired ResultsConstraint to use to select the nodes and edges to show in the current results.
5. Boolean flag that forces the whole network output if true, or not if false.

6.1 Developer Guide

You can extend this simulator adding new output formats. There are different steps for building new outputs, depending on your requirements. At the following lines you can see all possible steps:

1. Builds a new class that implements the **planet.commonapi.results.ResultsFactory** interface. By default, there is the **planet.generic.commonapi.results.ResultsFactoryImpl** implementation, with no constraints to use within any output format type. In the most of cases, this class is just the necessary one.
2. Builds a new class that implements the **planet.commonapi.results.ResultsEdge** interface. By default, the **planet.generic.commonapi.results.ResultsEdgeImpl** is just the necessary for the most of cases.
3. Builds a new class that implements the **planet.commonapi.results.ResultsConstraint** interface. By default is used the **planet.generic.commonapi.results.ResultsIdleConstraint** that includes all Nodes and edges into the generated output.
4. Builds a new class that implements the **planet.commonapi.results.ResultsGenerator** interface. This class is always mandatory implementation. This includes the schema of the output format really. See **planet.generic.commonapi.results.{ResultsGMLGenerator | ResultsPajekGenerator}** to take examples.
5. Builds a new class that implements the **planet.util.PropertiesInitializer** interface. This should contain all specific attributes for the new output format (colours, font type, font size, …).

Once these classes are implemented (or reused the existing ones), you need to specify them into the SCF to make it available from your desired test. You have to append the fully qualified class names into specific entries, using the **comma separated format**. Each position in the list of values for these entries is related to the same results type.

See an example from a SCF currently distributed with the simulator:

```
##########################################################################
# RESULTS PART
#
##########################################################################

#
# IMPORTANT: All different results attributes must appear in comma
# separated format, using each position for the same results type for
# all attributes.
#
```

```
########## OPTIONAL ATTRIBUTES: Test dependant

# The default ResultsFactory class
RESULTS_FACTORY =
      planet.generic.commonapi.results.ResultsFactoryImpl, \
      planet.generic.commonapi.results.ResultsFactoryImpl

# The default ResultsEdge class
RESULTS_EDGE =
      planet.generic.commonapi.results.ResultsEdgeImpl, \
      planet.generic.commonapi.results.ResultsEdgeImpl

# The default ResultsConstraint class
RESULTS_CONSTRAINT =
      planet.generic.commonapi.results.ResultsIdleConstraint, \
      planet.generic.commonapi.results.ResultsIdleConstraint

# The default ResultsGenerator class
RESULTS_GENERATOR =
      planet.generic.commonapi.results.ResultsGMLGenerator, \
      planet.generic.commonapi.results.ResultsPajekGenerator

# The default PropertiesInitializer for results properties
RESULTS_PROPERTIES =
      planet.generic.commonapi.results.ResultsGMLProperties, \
      planet.generic.commonapi.results.ResultsGMLProperties

# The unique names for each results type
RESULTS_UNIQUE_NAME =   GML, \
                        PAJEK
```

Figure 16. Example of results configuration into a SCF

As you can see, the results properties class name is the same for GML and Pajek outputs. The Pajek generator doesn't use any external property, but for compatibility within the simulator, it requires a valid class name.

At last, in the **`RESULTS_UNIQUE_NAME`** entry appears the unique names for the currently available results types. These names are used into the Java source code to get the required output results. The GML and Pajek names are included into the **planet.generic.commonapi.results.ResultsNames** class.

7 Future Work

There are some extensions of this simulator to design and to implement. We are agreed for any future collaboration on these terms. In the following list appears some of the future work:

7.1 TCP/IP Wrapper

This point should add the ability to run this simulator under a simulated or experimental environment. Currently only has been implemented the simulated environment, where whole network is built step by step (simulated time). The next step is to build the necessary architecture and elements into the simulator to be able to run the whole network under an experimental environment, using TCP or UDP communication between nodes.

Suggestions:
We have just included a new attribute on the SCFs with the name `SIMULATOR_ENVIRONMENT` to show the desired execution environment: SIMULATION (only this is available at now) and EXPERIMENTAL (for real TCP/IP communication).

7.2 Gathering Statistics

The current implementation is focused to get the maximum speedup on all simulation tests. But, other requirements are necessary in a research environment, as for example, the statistics. Number of connections currently available, number of incoming RouteMessages, number of outgoing RouteMessages, average of hops in RouteMessages delivering and so on is examples of statistics to gather on the most of tests.

Suggestions:
We believe that the AOP is a good candidate in this job. Because of all Node implementations are no variable, you should build some Aspects to intercept any required information and gather this data into the internal structure of Aspects. The benefits of this schema are the followings:

1. **No reprogramming is required**. Only a recompilation with or without aspects is needed to get a simulation with statistics or with the maximum speedup, respectively.
2. **No other cost is added**. When a simulation with the maximum speedup is needed, the data structure and process to gathering statistics doesn’t affect to the test.

8 Annexes

A. Chord implementation specification
B. Symphony implementation specification
C. Properties: configuration files specification

A. Chord implementation specification

The current implementation of Chord overlay is based on **[1]**, with the broadcast algorithm specified on **[2]**. The source code appears within packages **planet.chord** and **planet.chord.message**.

Data Structure

The most important data structure is related to the node connectivity. They are the **finger table** and the **successor list**.

The **finger table** has the same number of entries than number of bits of the Id (by default 32, in the available range of [32..160]). Each entry of this table represents a jump of 2^i where **i** goes in range [0..MAX_BITS-1] (by default between [0..31]).

The **successor list** is a collection of real successors of successors of each node, with a certain maximum number (by default 16).

Periodic tasks

There are two periodic tasks that force the stabilization of the node and review the finger table entries are correct. These two tasks update both data structures specified above: finger table and successor list. The rest of communication is on demand.

Brief description of Chord communication

There are some types of communication between Chord nodes. Any communication category as successor list specification, broadcast messages and so on are defined as **communication types**. For each type can appear different modalities of communication: one has to require an answer, other ones no response is required and so on. These modalities are defined as **communication modes**. So, to identify a RouteMessage in any communication is needed to specify its type and mode. See the following figure for more details:

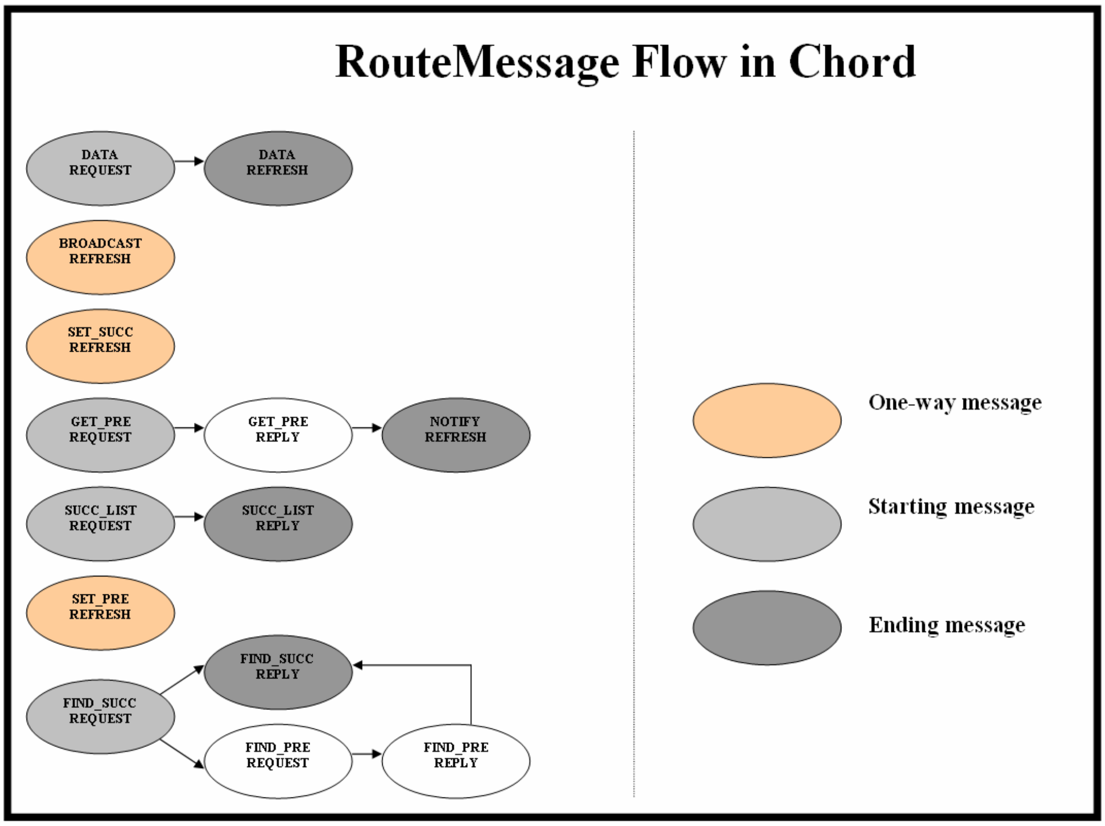

Figure 17. RouteMessage Flow in Chord

Description of communication objectives per type:

1. DATA: Type reserved to send any message of the Application level.
2. BROADCAST: Type reserved to send broadcast messages.
3. SET_SUCC: Shows the new successor to some node.
4. GET_PRE: Requests the predecessor of some key.
5. NOTIFY: Based on the GET_PRE response, send a NOTIFY RouteMessage to show that the local node is the predecessor of another one.
6. SUCC_LIST: Type reserved to maintain and inform the current successor list of the nodes.
7. SET_PRE: Type reserved to set the predecessor node from a node.
8. FIND_SUCC: Type reserved to find the successor of the local node.
9. FIND_PRE: Type reserved to find the immediate successor of any new incoming node to the network.

There are only three communication modes:

1. REFRESH: Shows that a RouteMessage is only one-way. No response is required.

2. REQUEST: This mode specifies that a response is required.
3. REPLY: This mode is the response for a REQUEST RouteMessage.

B. Symphony implementation specification

The current implementation of Symphony overlay is based on **[13]**. The source code appears within packages **planet.symphony**, **planet.symphony.behaviours** and **planet.symphony.messages**.

Data Structure

The most important data structure is related to the node connectivity. They are the **incoming list**, **outgoing list** and **neighbours set**.

The **incoming list** is the collection of incoming and locally accepted long distance connections from other nodes, with a maximum number of these connections.

The **outgoing list** is the collection of outgoing and remotely accepted long distance connections to other nodes, with a maximum number of these connections.

The **neighbours set** is the collection of neighbours of the current node. These neighbours are the successors and predecessors, with a maximum number of connections for the successors. Because of the ring topology of Symphony networks, also will attempt the same number of predecessors.

All these maximums are specified into the Symphony attributes into the SCF.

Periodic tasks

Only needs a periodic task that force the stabilization of the node and its neighbours. This task consists to send the full current neighbours set to the local node neighbours. With this operation, all nodes are updated with the nearer neighbours, and force closing farther neighbours connections.

On demand tasks

When the local estimation of the network size is increased up to 200% or is decreased down to 50%, a new task is started to fix another long distance connection. Its objective is to adapt the local long distance connections to the current network topology.

Brief description of Symphony communication

There are some types of communication between Symphony nodes. Any communication category is defined as **communication types**. For each type can appear different modalities of communication: one has to require an answer, other ones no response is required and so on. These modalities are defined as **communication modes**. So, to identify a RouteMessage in any communication is needed to specify its type and mode.

Description of communication objectives per type:

1. DATA: Type reserved to send any message of the Application level.
2. QUERY_JOIN: Message sent when a node joins to the network and queries for its immediate successor.
3. SET_INFO: Type reserved to send all current neighbours set to node neighbours for fixing them.
4. QUERY_CONNECT: Type reserved to request of a new long distance.
5. ACCEPT_CONNECT: Type reserved for a positive response of a QUERY_CONNECT.
6. CANCEL_CONNECT: Type reserved for a negative response of a QUERY_CONNECT.
7. CLOSE_LONG_CONNECT: Type reserved to force the closing of a long distance connection from a peer.
8. CLOSE_NEIGHBOUR_CONNECT: Type reserved to force the closing of a neighbour connection from a peer.

There are only three communication modes:

1. REFRESH: Shows that a RouteMessage is only one-way. No response is required.
2. REQUEST: This mode specifies that a response is required.
3. REPLY: This mode is the response for a REQUEST RouteMessage.

C. Properties: configuration files specification

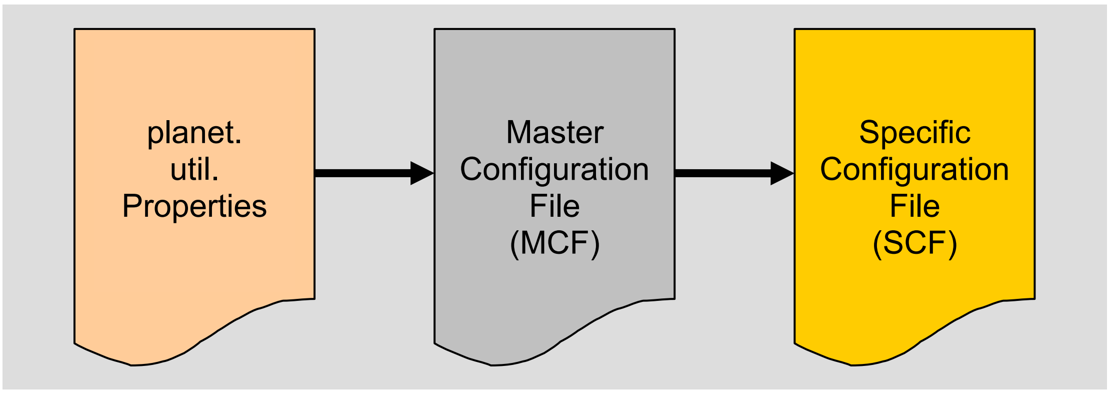

Figure 18. Properties loading process

This is the process followed by the planet.util.Properties to load all attributes from a SCF:

1. **planet.util.Properties** opens the **MCF** and search a **key** specified from the current test.
2. On the MCF appear all available entries for all currently distributed tests. The **key** refers to a specific test and the value is the path to the **SCF** required for that test. Once this path is get, planet.util.Properties opens this SCF.
3. The third step is the hardest. **All** required **attributes** have to be **loaded into** the own **planet.util.Properties**. It contains attributes to save the loaded values from the SCF. These values can be String, int or Class instances on the most of cases. When a Class is loaded, it verifies that the required interface or abstract class is implemented or extended.

C.1. Master Configuration File (MCF)

This is the currently distributed MCF:

```
######################################################################
# Main configuration file:                                           #
# ------------------------------                                     #
# This file specifies which real configuration file is used to run   #
# the desired test.                                                  #
#                                                                    #
# How to use:                                                        #
# ------------------------------                                     #
# By default, all keys of this file just specify the required        #
# configuration file for each test. You can modify their values at   #
# any moment to put any other filename.                              #
#                                                                    #
# If appears repeatedly the same key, you can select which is the    #
# current configuration for that test. How? Only you have to leave   #
# uncomment one of them.                                             #
#                                                                    #
# Remember that you can modify the values that appear in             #
# configuration files, too.                                          #
#                                                                    #
# Made by:                                                           #
#  Jordi Pujol Ahullo (jordi.pujol@estudiants.urv.es)                #
# Under:                                                             #
#  Planet Project: http://ants.etse.urv.es/planet                    #
#  PlanetSim:      htpp://ants.etse.urv.es/planetsim                 #
######################################################################

#
# The following properties specifies required configuration file for#
# each test.
#

######################################################################
# This test only runs under Chord overlay
#
IDTEST = ../conf/chord.properties

######################################################################
SIMNETTEST = ../conf/chord.properties
#SIMNETTEST = ../conf/symphony.properties

######################################################################
# This test only runs under Chord overlay
#
SIMPLETEST = ../conf/chord.properties

######################################################################
# This test only runs under Chord overlay
#
SIMTEST = ../conf/chord.properties

######################################################################
TESTPOOL = ../conf/chord.properties

######################################################################
# This test only runs under Chord overlay
#
BAD_SIMNETTEST = ../conf/chord.properties

######################################################################
# This test only runs under Chord overlay (because implements the
# broadcast)
#
BROADCAST_BROADCASTTEST = ../conf/chord_broadcast.properties


######################################################################
# This test only runs under Chord overlay
#
DHT_DHTTEST = ../conf/chord_dht.properties

######################################################################
#DHT2_DHTTEST = ../conf/chord_dht2.properties
DHT2_DHTTEST = ../conf/symphony_dht2.properties


######################################################################
###########
FACTORY_TESTAPPFACTORY = ../conf/chord.properties
```

```
#FACTORY_TESTAPPFACTORY = ../conf/symphony.properties
#FACTORY_TESTAPPFACTORY = ../conf/trivial.properties

#####################################################################
FACTORY_TESTENDPOINTFACTORY = ../conf/chord.properties
#FACTORY_TESTENDPOINTFACTORY = ../conf/symphony.properties
#FACTORY_TESTENDPOINTFACTORY = ../conf/trivial.properties

#####################################################################
FACTORY_TESTIDFACTORY = ../conf/chord.properties
#FACTORY_TESTIDFACTORY = ../conf/symphony.properties
#FACTORY_TESTIDFACTORY = ../conf/trivial.properties

#####################################################################
# This test only runs under Chord and Symphony overlays
# (The Trivial P2P requires specific stabilization process)
#
FACTORY_TESTNETFACTORY = ../conf/chord.properties
#FACTORY_TESTNETFACTORY = ../conf/symphony.properties

#####################################################################
FACTORY_TESTNODEFACTORY = ../conf/chord.properties
#FACTORY_TESTNODEFACTORY = ../conf/symphony.properties
#FACTORY_TESTNODEFACTORY = ../conf/trivial.properties

#####################################################################
FACTORY_TESTNODEHANDLEFACTORY = ../conf/chord.properties
#FACTORY_TESTNODEHANDLEFACTORY = ../conf/symphony.properties
#FACTORY_TESTNODEHANDLEFACTORY = ../conf/trivial.properties

#####################################################################
# This test only runs under Chord and Symphony overlays
# (The Trivial P2P requires specific stabilization process)
#
GML_GMLTOPOLOGY_GMLTOPOLOGYTEST = ../conf/chord.properties
#GML_GMLTOPOLOGY_GMLTOPOLOGYTEST = ../conf/symphony.properties

#####################################################################
# This test only runs under Chord and Symphony overlays
# (The Trivial P2P requires specific stabilization process)
#
HELLOWORLD_DHTPEERTEST = ../conf/chord.properties
#HELLOWORLD_DHTPEERTEST = ../conf/symphony.properties

#####################################################################
# This test only runs under Chord overlay
#
SCRIBE_SCRIBEPEERTEST = ../conf/chord_scribe.properties

#####################################################################
# This test only runs under Chord overlay
#
SCRIBE_SCRIBETEST = ../conf/chord_scribe.properties

#####################################################################
###########
# This test only runs under Chord overlay
#
SERIALIZE_GENSERIALIZEDFILE = ../conf/chord.properties
```

```
#####################################################################
# You have to ensure the match of serialized network with current
# configuration
#
SERIALIZE_LOADSERIALIZEDFILE = ../conf/chord_serialize.properties
#SERIALIZE_LOADSERIALIZEDFILE = ../conf/symphony_serialize.properties

#####################################################################
# This test only runs under Chord and Symphony overlays
# (The Trivial P2P requires specific stabilization process)
#
SERIALIZE_SERIALIZENETWORK = ../conf/chord.properties
#SERIALIZE_SERIALIZENETWORK = ../conf/symphony.properties

#####################################################################
TRIVIALP2PTEST_TRIVIALTEST = ../conf/trivial.properties
```

Figure 19. PLANETSIM/conf/master.properties MCF

This file structure offers the ability to run any test without to modify any file other file, by default. It associates a SCF to each test.

If there are more than one available SCF for one test, it appears one key uncommented and the other ones commented (with '#' at the beginning). You can choose the SCF to use at any moment leaving uncomment only the desired line. On these cases, each line relates an available overlay.

If only there is a line for a test, it is only the overlay available and you have not to force it with other ones.

C.2. Specific Configuration File (SCF)

A SCF relates one overlay with one or more tests. So, each SCF contains the values required for that overlay and correct values for those tests. To modify any value you have to know its effects and other possible values.

We show a SCF for the Symphony overlay because it is a good example:

```
#####################################################################
# Chord configuration file:                                         #
# ----------------------------                                      #
# This file specifies all properties (including the Chord specifics #
# ones) to run any test with the Chord overlay.                     #
#                                                                   #
# How to use:                                                       #
# ----------------------------                                      #
# All properties are divided into different semantical parts.       #
# You must specify the desired properties values into the following #
# lines.                                                            #
#                                                                   #
# Made by:                                                          #
#  Jordi Pujol Ahullo (jordi.pujol@estudiants.urv.es)                #
# Under:                                                             #
#  Planet Project: http://ants.etse.urv.es/planet                    #
#  PlanetSim:      htpp://ants.etse.urv.es/planetsim                 #
######################################################################


######################################################################
# FACTORIES PART                                                     #
######################################################################

########## MANDATORY ATTRIBUTES

# The default NetworkFactory class
FACTORIES_NETWORKFACTORY =
planet.generic.commonapi.factory.NetworkFactoryImpl

# The default IdFactory class
FACTORIES_IDFACTORY = planet.generic.commonapi.factory.IdFactoryImpl

# The default NodeHandleFactory class
FACTORIES_NODEHANDLEFACTORY =
planet.generic.commonapi.factory.NodeHandleFactoryImpl

# The default NodeFactory class
FACTORIES_NODEFACTORY =
planet.generic.commonapi.factory.NodeFactoryImpl

# The default RouteMessagePool class
FACTORIES_ROUTEMESSAGEPOOL =
planet.generic.commonapi.factory.RouteMessagePoolImpl

# The default Network class
FACTORIES_NETWORK = planet.generic.commonapi.NetworkImpl

# The default NodeHandle class
FACTORIES_NODEHANDLE = planet.generic.commonapi.NodeHandleImpl

# The default RouteMessage class
FACTORIES_ROUTEMESSAGE = planet.generic.commonapi.RouteMessageImpl

# The default network topology.
# Default possible values: RANDOM |CIRCULAR | SERIALIZED
FACTORIES_NETWORKTOPOLOGY = RANDOM

# The default initial network size
FACTORIES_NETWORKSIZE = 1000


########## OPTIONAL ATTRIBUTES: Test dependant

# The default ApplicationFactory class
FACTORIES_APPLICATIONFACTORY =
planet.generic.commonapi.factory.ApplicationFactoryImpl

# The default EndPointFactory class
FACTORIES_ENDPOINTFACTORY =
planet.generic.commonapi.factory.EndPointFactoryImpl

# The default Application class
FACTORIES_APPLICATION = planet.test.helloworld.DHTApplication
```

```
# The default EndPoint class
FACTORIES_ENDPOINT = planet.generic.commonapi.EndPointImpl


#####################################################################
# SIMULATOR PART                                                    #
#####################################################################

########## MANDATORY ATTRIBUTES

# The number of stabilization steps for any node at join or leave
# Default value: 2
SIMULATOR_SIMULATION_STEPS = 2

# The log level (to use by Logger.log(...) )
# Default possible values (from more to less important logs): 0
# (error), 1 (events), 2 (node info), 3 (message)
SIMULATOR_LOG_LEVEL = 0

# The print level for whole network (to use by
# GenericApp.printNetwork() method)
# Default possible values: 0 (no print), 1 (pretty print), 2 (full
# print)
SIMULATOR_PRINT_LEVEL = 2

# The environment for the current simulation
# Default possible values: SIMULATION (by steps), EXPERIMENTAL (by
# threads and real TCP connections)
# Only SIMULATION has available
SIMULATOR_ENVIRONMENT = SIMULATION

# The queue size for the incomming and outgoing queues
# Default value: 128
SIMULATOR_QUEUE_SIZE = 128

# The maximum number of messages to be processed per node per step
# Default value: 128
SIMULATOR_PROCESSED_MESSAGES = 128

########## OPTIONAL ATTRIBUTES: Test dependant

# The events filename to load
SIMULATOR_EVENT_FILE =


#####################################################################
# SERIALIZATION PART                                                #
#####################################################################

########## OPTIONAL ATTRIBUTES: Test dependant

# Serialized file that contains the network to be loaded
SERIALIZATION_INPUT_FILE = network.psim

# Filename to which serialize the final state
SERIALIZATION_OUTPUT_FILE = network.psim

# Identifies if the output file must be replaced with new outputs,
# when the state is serialized
SERIALIZATION_REPLACE_OUTPUT_FILE = false
```

```
######################################################################
# BEHAVIOURS PART                                                    #
######################################################################

########## OPTIONAL ATTRIBUTES: Overlay dependant

# The default BehaviourFactory class
BEHAVIOURS_FACTORY =
planet.generic.commonapi.behaviours.BehavioursFactoryImpl

# The default BehavioursPool class
BEHAVIOURS_POOL =
planet.generic.commonapi.behaviours.BehavioursPoolImpl

# The default BehavioursRoleSelector class
BEHAVIOURS_ROLESELECTOR =
planet.generic.commonapi.behaviours.BehavioursRoleSelectorImpl

# The default BehavioursInvoker class
BEHAVIOURS_INVOKER =
planet.generic.commonapi.behaviours.BehavioursInvokerImpl

# The default BehavioursFilter class
BEHAVIOURS_FILTER =
planet.generic.commonapi.behaviours.BehavioursIdleFilter

# The default BehavioursPattern class
BEHAVIOURS_PATTERN =
planet.generic.commonapi.behaviours.BehavioursPatternImpl

# The default PropertiesInitializaer class for the behaviours
# properties
BEHAVIOURS_PROPERTIES =
planet.generic.commonapi.behaviours.BehavioursPropertiesImpl

# The default number of message types used in the current overlay
# Default value for Symphony: 8
BEHAVIOURS_NUMBEROFTYPES = 8

# The default number of message modes used in the current overlay
# Default value for Symphony: 3
BEHAVIOURS_NUMBEROFMODES = 3

######################################################################
# SPECIFIC PROPERTIES OF BEHAVIOURS PART                             #
######################################################################

########## OPTIONAL ATTRIBUTES: Overlay dependant

# The default percentage of faulty nodes
# Default possible values: [0..100]
BEHAVIOURS_PROPERTIES_FAULTY_NODES = 0

# The default distribution of malicious node
# Default possible values: UNIFORM | CHAIN
BEHAVIOURS_PROPERTIES_MALICIOUS_DISTRIBUTION = CHAIN

# Identifies when to show specific debug info for behaviours
BEHAVIOURS_PROPERTIES_DEBUG = false
```

```
# NOTE: The following keys start by 'BEHAVIOURS_PROPERTIES_INSTANCE',
# ended with an incremental integer number to make them different

# All required instances for the current behaviours implementation
# COLUMN NAMES:
MESSAGE                 MESSAGE
# UNIQUE NAME                   =             BEHAVIOUR CLASS
,   TYPE                 ,  MODE  , PROBABILITY , LOCALITY , ROLE
#----------------------------------------------------------------------
----------------------------------------------------------------------
------------------------
BEHAVIOURS_PROPERTIES_INSTANCE_1 =
planet.symphony.behaviours.RoutingBehaviour,               ?,
*,       1.0,          REMOTE,    NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_2 =
planet.symphony.behaviours.QueryJoinBehaviour,
QUERY_JOIN,              REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_3 =
planet.symphony.behaviours.SetInfoBehaviour,               SET_INFO,
REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_4 =
planet.symphony.behaviours.QueryConnectBehaviour,
QUERY_CONNECT,           REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_5 =
planet.symphony.behaviours.AcceptConnectBehaviour,
ACCEPT_CONNECT,          REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_6 =
planet.symphony.behaviours.CancelConnectBehaviour,
CANCEL_CONNECT,          REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_7 =
planet.symphony.behaviours.CloseLongConnectBehaviour,
CLOSE_LONG_CONNECT,      REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_8 =
planet.symphony.behaviours.CloseNeighbourConnectBehaviour,
CLOSE_NEIGHBOUR_CONNECT, REFRESH, 1.0,          LOCAL,     NEUTRAL
BEHAVIOURS_PROPERTIES_INSTANCE_9 =
planet.symphony.behaviours.DataBehaviour,                  DATA,
*,       1.0,          ALWAYS,    NEUTRAL

# Default possible values:
# BEHAVIOUR CLASS, MESSAGE TYPE, MESSAGE MODE ==> Overlay dependant
# PROBABILITY ==> [0.0 .. 1.0] (never ..  always)
# LOCALITY    ==> LOCAL | REMOTE | ALWAYS
# ROLE        ==> GOOD  | BAD    | NEUTRAL


######################################################################
# OVERLAY PART                                                       #
######################################################################

########## MANDATORY ATTRIBUTES

# The default Id class
OVERLAY_ID = planet.symphony.SymphonyId

# The default Node class
OVERLAY_NODE = planet.symphony.SymphonyNode

# The default OverlayProperties implementation class
OVERLAY_PROPERTIES = planet.symphony.SymphonyProperties
```

```
# Identifies if this overlay implementation uses behaviours
# Default possible values: false | true
OVERLAY_WITH_BEHAVIOURS = true

#####################################################################
# SYMPHONY SPECIFIC PART                                            #
#####################################################################

########## MANDATORY ATTRIBUTES

# The default number of long distance connections
# Default value: 2
SYMPHONY_MAX_LONG_DISTANCE = 2

# The default maximum number of members in successor list
# Default value: 2
SYMPHONY_MAX_SUCCESSOR_LIST = 2

# The default maximum number of retries to obtain a connection
# to the same long distance node
# Default value: 3
SYMPHONY_MAX_RETRIES_NEW_LONG_DISTANCE = 3

# The default maximum number of retries to enter to the network by
# the same bootstrap
# Default value: 10
SYMPHONY_MAX_JOIN_RETRIES = 10

# The default number of stabilize steps
# Default value: 3
SYMPHONY_STABILIZATION_STEPS = 3

#####################################################################
# RESULTS PART                                                      #
#####################################################################

#
# IMPORTANT: All different results attributes must appear in comma
# separated format, using each position for the same results type for
# all attributes.
#

########## OPTIONAL ATTRIBUTES: Test dependant

# The default ResultsFactory class
RESULTS_FACTORY =
      planet.generic.commonapi.results.ResultsFactoryImpl, \
      planet.generic.commonapi.results.ResultsFactoryImpl

# The default ResultsEdge class
RESULTS_EDGE =
      planet.generic.commonapi.results.ResultsEdgeImpl, \
      planet.generic.commonapi.results.ResultsEdgeImpl

# The default ResultsConstraint class
RESULTS_CONSTRAINT =
      planet.generic.commonapi.results.ResultsIdleConstraint, \
      planet.generic.commonapi.results.ResultsIdleConstraint

# The default ResultsGenerator class
```

```
RESULTS_GENERATOR =
      planet.generic.commonapi.results.ResultsGMLGenerator, \
      planet.generic.commonapi.results.ResultsPajekGenerator

# The default PropertiesInitializer for results properties
RESULTS_PROPERTIES =
      planet.generic.commonapi.results.ResultsGMLProperties, \
      planet.generic.commonapi.results.ResultsGMLProperties

# The unique names for each results type
RESULTS_UNIQUE_NAME =   GML, \
                        PAJEK

#####################################################################
# GML SPECIFIC RESULTS PART                                         #
#####################################################################

########## OPTIONAL ATTRIBUTES: Test dependant

# The default width of the virual bounding box
RESULTS_PROPERTIES_GML_WIDTH = 20.0f

# The default height of the virtual bounding box
RESULTS_PROPERTIES_GML_HEIGHT = 20.0f

# The default shape of the node
RESULTS_PROPERTIES_GML_SHAPE = ellipse

# The default fill color for the shape of the node (in #'RRGGBB'
# format)
RESULTS_PROPERTIES_GML_FILL = CCCCFF

# The default alternative fill color for the shape of the node (in
#'RRGGBB' format)
RESULTS_PROPERTIES_GML_ALTERNATIVE_FILL = 00FF66

# The default color of the border line (in #'RRGGBB' format)
RESULTS_PROPERTIES_GML_OUTLINE = 000000

# The default font size of the node Id lavel
RESULTS_PROPERTIES_GML_FONT_SIZE = 12

# The default font name of the node Id label
RESULTS_PROPERTIES_GML_FONT_NAME = dialog

# The default minimal node distance arranged on a circle
RESULTS_PROPERTIES_GML_MINIMAL_NODE_DISTANCE = 50
```

Figure 20. PLANETSIM/conf/symphony.properties SCF

This is one SCF for the Symphony overlay. As you can see, there are different parts. But, there are some ones that are mandatory. They are the following:

1. FACTORIES PART: On it appears all basic classes to be loaded by the simulator, including the factory classes (following the Factory Method design pattern) and all the instance to be built by these factories. The last lines are

related to the Application level, and only will be loaded on demand.

2. SIMULATOR PART: These attributes are related to the basic simulator operation, like size of internal queues for incoming and outgoing RouteMessages and so on. The last line are related to the events filename, and only will be loaded on demand.
3. OVERLAY PART: It defines all classes related to the overlay to be loaded: Node, Id and OverlayProperties. The last line shows when the current overlay uses behaviours. When true, the optional BEHAVIOURS PART is loaded automatically.

The rest of the parts are test or overlay dependant. See below for their descriptions (in order of appearance):

1. SERIALIZATION PART: These attributes show the file with the serialized network to be loaded and the path where save the current state.
2. BEHAVIOURS PART: This part is automatically loaded when the overlay uses behaviours, and specifies all classes to use for the behaviours architecture.
3. SPECIFIC PROPERTIES OF BEHAVIOURS PART: These attributes are overlay dependant and also are loaded when the overlay uses behaviours.
4. SYMPHONY SPECIFIC PART: Each overlay can specify its own attributes. In this example is Symphony.
5. RESULTS PART: This part is test dependant and offers the ability of write outputs of specified types. Contains all required classes to build the output and no blank can appear.
6. GML SPECIFIC RESULTS PART: When a result type needs some specific attributes, they also have to appear on this document. In the example, GML attributes are specified.

9 References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", *ACM SIGCOMM 2001*, San Diego, CA, pp. 149-160, August 2001

[2] El-Ansaray, S.; Alima, L.O.; Brand, P.; et al. "Efficient Broadcast in Structured P2P Networks", *2nd International Workshop on Peer-to-Peer Systems, IPTPS'03*, Berkeley, CA, Febraury 2003

[3] Castro, M., Druschel, P., et al, "Scalable Application-level Anycast for Highly Dynamic Groups", *Proc. of NGC'03,* September 2003.

[4] Dabek, F., Zhao, B.Y., Druschel, P., Kubiatowicz, J., and Stoica I., "Towards a Common API for Structured Peer-to-Peer Overlays", *2nd International Workshop on Peer-to-Peer Systems, IPTPS'03,* Berkeley, CA, February 2003.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] PeerSim Peer-to-Peer Simulator. http://peersim.sourceforge.net/

[10] 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.

[11] 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.

[12] 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.

[13] Singh, G.M., Bawa, M., Raghavan, P. “Symphony: Distributed Hashing in a Small World”. *Proceedings of USITS’03*, Seattle, WA.

[14] The Network Simulator – ns – 2.
http://www.isi.edu/nsnam/ns/

[15] J-Sim. http://www.j-sim.org/

[16] MACEDON. http://macedon.ucsd.edu/

[17] The GML file format:
http://www.infosun.fmi.uni-passau.de/Graphlet/GML/

[18] yEd – Java ™ Graph Editor
http://www.yworks.com/en/products_yed_about.htm

[19] Pajek. http://vlado.fmf.uni-lj.si/pub/networks/pajek/