BISON
IST-2001-38923

Biology-Inspired techniques for Self Organization in dynamic Networks

Simulation Environment

Deliverable Number:	D12-D13
Delivery Date:	June 2004 (Revised January 2005)
Classification:	Public
Contact Authors:	Alberto Montresor, Gianni Di Caro, Frederick Ducatelle, Gian Paolo Jesi
Document Version:	Final (February 25, 2005)
Contract Start Date:	1 January 2003
Duration:	36 months
Project Coordinator:	Università di Bologna (Italy)
Partners:	Telenor ASA (Norway), Technische Universität Dresden (Germany), IDSIA (Switzerland)

Project funded by the European Commission under the Information Society Technologies Programme of the 5th Framework (1998-2002)

Abstract

This document constitutes the accompanying documentation for the prototype Deliverables D12 and D13. As described in the technical annex of project BISON, these deliverables were meant to provide a generic simulation environment for dynamic networks (in D12), and two different customizations of the environment for ad-hoc networks and P2P/overlay networks (in D13). As described in Deliverable D11, however, we have taken a different route. The simulation architecture for ad-hoc networks builds up on an existing packet-level simulator, namely *QualNet*. On the other hand, the architecture of the overlay network simulator is based on a completely custom design specifically optimized for the simulation of very large networks and the analysis of graph-theoretical structures. This document integrates Deliverable D11 by describing the implementation of these simulation environments, with particular emphasis on our home-grown simulation environment for overlay networks: PEERSIM.

Contents

1 Introduction

The production of a simulation environment that supports the design, the analysis and the evaluation of biology-inspired techniques in ad-hoc networks and overlay systems is one of the objectives of BISON. In the original workplan contained in the technical annex, the design and development of such framework have been subdivided in three distinct deliverables:

1. Deliverable D11 (completed at month 6) was mainly concerned with the design of the architecture for our simulation environment. Originally, such environment was aimed at providing a *general purpose* "testbed" for studying and experimenting with biology-inspired algorithms, in order to understand their properties and evaluate their performance. The main requirement of such environment was the following:

   > *... its ability to reproduce the distinctive properties and features that are common to all kinds of dynamic networks. Ad-hoc networks, P2P and Grid computing are all described by events that occur on a graph, composed of nodes connected by edges*.

2. Deliverable D12 was meant to provide a *generic* prototype for the architecture designed in Deliverable D11, to be customized later in deliverable D13.

3. Finally, the task of Deliverable D13 is to augment the generic environment implemented in Deliverable D12 with specialized components for the particular dynamic networks (ad-hoc networks, p2p/overlay networks) and application areas to be simulated.

Based on our first-year studies, we have taken a different route and decided to use a commercial simulator (QualNet [6]) for the ad-hoc network part and to build in-house a special-purpose simulator (PEERSIM [4]) for the p2p/overlay network part. This decision has been motivated and described in the 6-month deliverable D11 and explained during the review meeting in Bruxelles. The implication of this decision is that Deliverables D12 and D13 merge into a single Deliverable, since it does not make sense anymore to be presenting the simulation environment as two separate ("general purpose" and "specialized") components. This deliverable is thus concentrated on the simulation environment

The rest of document is organized as follows. Section 2 describes of the simulation environment for ad-hoc networks. That is, it describes the general characteristics of QualNet. Section 3 illustrates the implementation of PEERSIM, the open-source simulator that we have developed for the evaluation of application-level protocols and algorithms for overlay networks. In Section 4 we report about ongoing work aimed at providing empirical validation of PEERSIM, by implementing some of our algorithms in the Planet-Lab testbed.

2 Simulation Environment for Ad-hoc Networks

In the light of the review of the candidate network simulators performed in Deliverable D11, we have decided to use *QualNet* as the environment for development and simulation. None of the reviewed simulators seem to really possess the whole set of characteristics required by BISON's research plans and objectives. However, QualNet appears as the best compromise

in terms of number of pre-built components, modularity, scalability, and modifiability. In this sense, we see QualNet as an effective simulation framework on top of which we can build the complete BISON's simulation architecture for mobile ad hoc networks, by adding new models and protocols, and by adapting/modifying the existing QualNet's components to BISON's specific needs.

QualNet is a commercial network simulation package composed of several tools for: (i) a graphical experiment design and animation, (ii) heterogeneous network simulation, (iii) graphical protocol design, (iv) graphical statistical analysis, (v) graphical packet tracing, and (vi) graphical network data collection. Graphical interfaces are based on Java and XML, while models and protocols are written in ANSI C. QualNet is available on both Windows and Linux platforms and we could use it on both platforms (Windows XP and Linux Red Hat 9.0) without the need for any modification at our code when migrating from one operating system to the other.

2.1 General architecture of QualNet

QualNet employs a layered architecture similar to that of the TCP/IP network protocol stack. As such, data traverses between adjacent layers. QualNet's protocol stack consists of, from top to bottom, the application, transport, network, link (MAC) and physical layer. Well-defined APIs are used to communicate between adjacent layers in protocol stack. As a general rule, layer communication can only occur between adjacent layers. However, this rule is not strictly enforced and may be broken by the developer if necessary. The `Animator` tool in QualNet provides a graphical visualization of the run-time activities of the nodes with the possibility to visualize also the activities at the the different layers (e.g., node mobility, actual radio range, active sessions, queue length, packet sending). Figure 1 shows two screen shots from `Animator`.

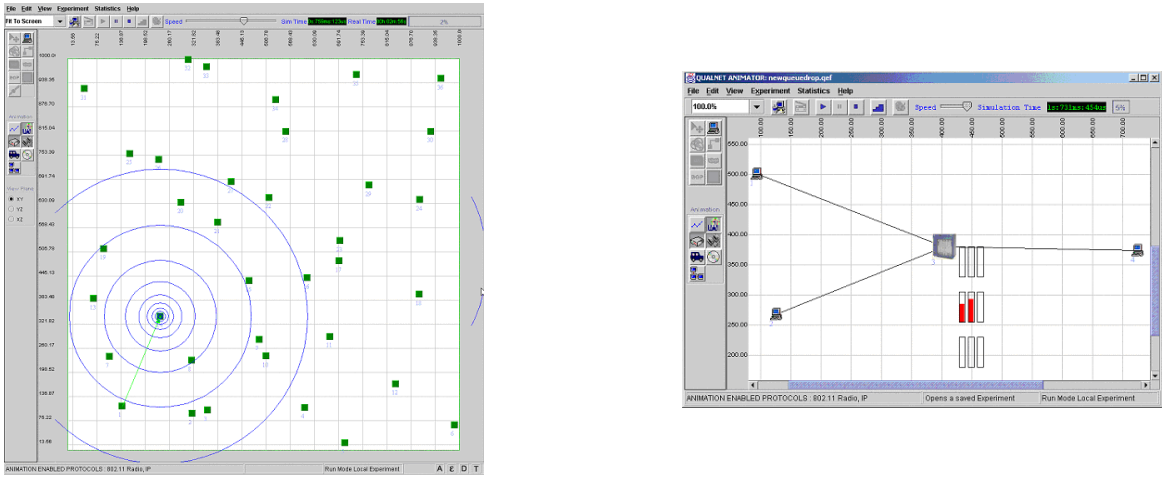

Figure 1: Two QualNet screen shots that show the graphical capabilities of the `Animator` tool. The left image shows the radio range and the fact that there is a packet being sent between the two nodes connected by the arrow. The right image shows the state of the different queues at a router.

We implemented our routing algorithms as *network layer protocols*. The network layer is responsible for data forwarding and queuing/scheduling. The Internet Protocol (IP) resides at this layer and is responsible for packet forwarding. Routing protocols can be implemented at this layer in order to manage/update the routing tables used by the IP for data forwarding.

Protocols in QualNet essentially operate as a *finite state machine* that changes state only on the occurrence of an event. The layer code (protocols), therefore, is implemented as an event handler, that receives an event data structure, called a *message*, containing the type of event and the associated data. The event handler then parses the data and handles it appropriately, perhaps scheduling further events at the next layer or at the same layer. The `Designer` tool in QualNet helps to design a protocol in the form of a finite state machine by using a graphical interface. The visual representation of the states and transitions is based on the UML state diagram notation. States can be created and placed on the diagram. They can also be connected with transitions. States are storage houses for code that is executed whenever the state is entered. *ANSI C is used as programming language* to write state code. A state transition can be one of the following: (i) triggered, that happens upon the arrival of a particular event, the "trigger", (ii) guarded, that occurs when a particular Boolean condition, the guard, is true, (iii) automatic, that happens automatically upon completion of the state's entry code, and also (iv) triggered and guarded. Using the graphical interface of the `Designer` tool, only the code that has to be executed inside the states has to be explicitly written: the full C code for the protocol in order to integrate it into QualNet is automatically generated. Clearly, also the specifications for the name of the protocol, its operational layer, and all the state transitions have also to be made explicit but the task is made rather simple with the help of the graphical interface. Figure 2 shows the use of the `Designer` tool to connect the `Initial state` to a subsequent state by a state transition.

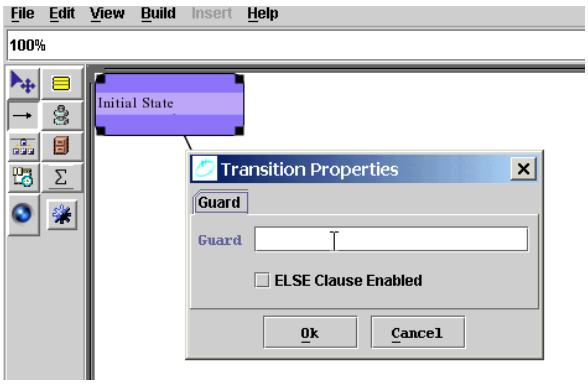

Figure 2: Example of use of the `Designer` tool to connect the `Initial state` to another state by a guarded state transition.

Figure 3 shows the general functioning of a typical QualNet protocol in state diagram form:

1. The protocol begins with an `Initial state` that initializes the protocol. The code of the initialization state reads external input to configure the state of the protocol and possibly activates timer events for periodic tasks.

2. The control is then passed to an event dispatcher.

3. When an event comes in to the protocol's layer, QualNet Simulator determines to which specific protocol it should be directed, and calls the event dispatcher for that protocol.
4. The event dispatcher determines what type of event it is, and calls an appropriate event handler to process it.
5. At the end of the simulation, a `Finalization function` is called for every protocol (at each node), to print out the collected statistics.

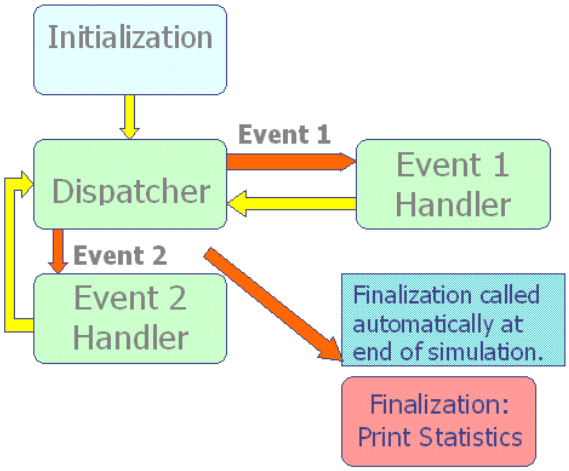

Figure 3: General behavior of a typical QualNet protocol in state diagram form.

Therefore, every protocol must implement three functions to interface to the Simulator: Initialization, Event dispatcher, and Finalization. When a protocol is written with the help of the `Designer` tool, the actual file which is written through the graphical interface is an XML file contained in the directory `gui/models` that is also the repository for both the C code and the directives for the graphical display of the finite state machine.

2.2 Models specific for mobile ad hoc networks

In this subsection we briefly describe the characteristics of the models that inside QualNet implement the management and simulation of components that are of specific interest for mobile ad hoc networks. In particular, we discuss the models for node mobility, and for the physical, MAC, transport and application layers. Each model has a set of parameters whose values can be assigned at running time.

Physical layer

QualNet provides a number of *radio models* in order to simulate the hardware that performs channel sensing, radio transmissions, receptions, and battery usage. These models and their parameters are not specified on a per-channel basis, but rather using node-specific and network-specific parameterization. The radio model for MANETs available in the standard version of QualNet is a *WaveLAN II / IEEE 802.11a/b* compatible radio device model [1]. *Omnidirectional*

and directional/steerable antenna models are available, and each node can have one or *multiple communication channels*. The radio signal *fading models* available in QualNet are narrowband flat fading models which implement the *Rayleigh* and the *Ricean* wireless physical layer model for fading. The fading model can be specified per channel. QualNet provides two models For pathloss radio propagation QualNet provides two models, the so-called *two-ray* and *free-space*. The two-ray one is the more realistic one.

Several parameters can be assigned in order to set the characteristics of the radio transmission/propagation and in particular of the effective *radio range*. Another component of the radio model which is relevant in case of using a bandwidth greater than 2 Mbps is the *fallback* behavior. It allows to have multirate transfer capabilities to perform dynamic rate switching.

MAC layer

QualNet provides a faithful implementation of a large number of MAC layer protocols both wired and wireless (802.11, CSMA, TDMA, SATCOM, MACA, ALOHA). The mostly used in MANETs is the *802.11a/b* [1].

Transport and application layers

At the transport layer QualNet comes with UDP and TCP, the two protocols of the TCP/IP suite, and with RSVP, which is used in QualNet as a control protocol for quality of service. In terms of applications, QualNet offers a wide range of possibilities (e.g., CBR, VBR, FTP, HTTP, Telnet, etc.).

Node mobility

In QualNet, mobility components work together with node distribution and terrain components to model nodes' mobility behavior. QualNet provides: (i) *Random waypoint* and trace files as mobility models, (ii) uniform, random, grid and file-based distribution for initial node placement (nodes can be also placed by hand through the graphical interface), and (iii) DTED (Digital Terrain Elevation Data) and DEM (Digital Elevation Model) for terrain description. The mobility model specifies an array of destinations and arrival times for each node. Each node moves straight towards the next destination with constant velocity, and QualNet calculates all the intermediate positions. In this way any trajectory can be followed in piecewise way. Elevation is determined by terrain data. Mobility can also include orientation.

3 Simulation Environment for Overlay Networks

PEERSIM has been developed with extreme scalability and support for dynamism in mind. It is composed of many simple extendable and pluggable components, with a flexible configuration mechanism. To allow for scalability and focus on self-organization properties of large scale systems, some simplifying assumptions have been made, such as ignoring the details of the transport communication protocol stack.

PEERSIM is distributed under the GNU Lesser General Public License version 2 (LGPL). Its source code is hosted at Sourceforge, the well-known repository for open-source projects. The home page of the project is located at `http://peersim.sourceforge.net/`, and contains the following material:

- The tarball file containing the source code of the project;
- The API documentation generated through Javadoc;
- A tutorial that describes how to implement simple P2P protocols for load-balancing, and how to run the corresponding experiments with PEERSIM. The focus is on the configuration mechanism; several configuration files are provided, illustrating how to compose several different protocols like aggregation [3, 5] and NEWSCAST [2].
- A tutorial that describes how to implement a topology generator with PEERSIM.

PEERSIM is written in the Java language. This choice is motivated by the fact that the reflection mechanism of Java enables the construction of complex simulations based on text-based configuration files.

In the following, we provide a general description of PEERSIM and its components. This description is a revision of the architectural design contained in Deliverable D11, augmented with implementation details.

The main goal of PEERSIM is to enable the composition of complex simulators without incurring in excessive overhead both in terms of memory and time. In order to achieve this goal, its architecture is as light-weight as possible. The idea is to provide a configuration service that enables the construction of simulators by "juxtaposing" together pre-existing or newly-developed components. Every component of the simulator is completely interchangeable, in order to allow developers to write their customized and optimized version when needed. Components are specified by object-oriented programmatic interfaces, whose methods describes the expected behavior of a component.

The basic building block of our simulation environment is the *configuration manager*. This module will be present in each simulation, and its task is to read configuration files and/or command-line parameters, and compose a simulation starting from the classes listed in these configuration specifications. The flexibility offered by this mechanisms enables developers to re-implement, when needed, every component of the system, with the freedom of re-using existing components for fast prototyping.

Figure 4 contains an example of configuration file. The example is neither complete nor exhaustive; it is meant to give an idea of the potentiality of the configuration manager. The components listed in the file represent some of the main elements that are used in our simulators.

The configuration manager is quite flexible, allowing the definition of constants (like `SIZE` and `DEGREE`) and the use of expressions (like the `SIZE/100`). Classes to be used in the simulation can be specified in two forms: either with the full-qualified name (like `peersim.core.IdleProtocol`), or simply by class name (like `DynamicNetwork`. In case of conflicts, the configuration manager stops reporting the problem and specifying the reasons for the conflict.

```
simulation.cycles 30
simulation.shuffle
random.seed 1234567890

SIZE 50000
DEGREE 20

overlay.size SIZE

protocol.0 peersim.core.IdleProtocol
protocol.0.degree DEGREE

protocol.1 example.aggregation.AverageFunction
protocol.1.linkable 0

init.0 peersim.dynamics.WireRegularRandom
init.0.protocol 0
init.0.degree DEGREE

init.1 example.aggregation.PeakDistributionInitializer
init.1.value SIZE
init.1.protocol 1

dynamics.0 DynamicNetwork
dynamics.0.add SIZE/100
dynamics.0.substitute

observer.0 example.aggregation.AverageObserver
observer.0.protocol 1
```

Figure 4: Example of configuration file for a simple simulation

Each simulation is driven by a *simulation engine* over a simulated *network*. The characteristics of the simulation engine and the network to be used in the simulation are described in the first lines of the configuration file.

The main elements that compose the dynamic behavior of the simulation are *protocols*, *initializers*, *dynamics* and *observers*. Each component is responsible for loading the appropriate components needed for its execution; for example, a simulation engine instantiates a network, the network instantiates a set of nodes, each nodes instantiates the appropriate protocols, and so on.

The flexibility of our simulation environment is twofold: neither the set of interfaces nor the set of implementations of those interfaces are fixed:

- For example, it is possible for a developer to decide that a certain component should be organized as a collection of sub-components, and decide the interfaces for these sub-components. For example, a dynamic network can be composed by a data structure for maintaining the network information, and an ”uptime” model that actually decides when a new node should join or an existing node should leave. This interface flexibility must

be expressed programmatically, in the sense that the developer will have to write new code to support it.

- On the other hand, developers are enabled to specify the actual implementation to be used in the simulator. For example, in the example, the average aggregation protocol (`AverageFunction`) is selected; it is possible to declaratively modify the simulation by simply changing the configuration file (provided that an appropriate implementation is available, otherwise a new implementation will have to be written).

Due to this flexibility, it is clear that our simulation environment will grow over time. Thus, the set of components listed in Figure 4 is very limited and represent only a first approximation of the final model. In the next sections we describe them, and we provide some ideas about how to extend them.

For a complete list of the available classes, please refer to the Javadoc documentation in the Peersim web site.

3.1 Simulation Engine

Two simulation engines are available in Peersim:

- In the *cycle-driven* simulator, simulation proceeds through time steps called *cycles*, in which all the nodes get a chance to execute. This enables the fast prototyping of promising protocols, and the analysis of the influence of different topologies on the studied functions.

- The *event-driven* simulation engine allows the execution of full-fledged protocols. Several kinds of transport substrates are available, some of them based on well-known traces derived by existing overlay networks.

In this particular configuration file, the first line tells the system to use the cycle-based simulation engine, and to run it for 30 cycles. At each cycle, the order in which nodes are executed is shuffled (as specified by the second line).

3.2 Network Topology

As described in deliverable D01, overlay topologies and overlay topology management will be a fundamental research topics in BISON. For this reason, the concept of topology constitutes the basic building block of all simulations. Its basic description is based on the concept of *network*, composed of a set of *nodes*. Each node n is associated with a non-empty list of *connections*, i.e. neighbors nodes that are known to n. Two basic operations are possible: to modify the set of connections, and to obtain the set of connections. Connections will be used to send messages over them, or to directly invoke methods on the connected components.

3.3 Protocols

The domain of protocols is clearly the most open one; several different functions and algorithms will be implemented on our simulator. So far, we have implemented some basic prototypes for aggregation, load balancing, superpeer topology construction, ranking, and generic topology management. Since the protocols that can be implemented are out of the scope of the simulation environment, we will not discuss them any further. In the example of 4, two protocols are configured: the `IdleProtocol`, whose task is to maintain a static connected topology, and `AverageFunction`, an aggregation protocol whose task is to compute an aggregate function (the average) over a set of values maintained by peers participating in the computation.

3.4 Initializers

Initializers are run at the beginning of the simulation in order to initialize the state of the system. In the specific example, the two initializers create a static regular random topology using the idle protocol (protocol identifier 0) and initialize the values to be aggregated using a peak distribution (protocol identifier 1).

3.5 Dynamics

Dynamics are run during the simulation, and modify the state of the system according to environmental inputs (for example, by simulating the crash of some nodes or by modifying the value to be aggregated. In the specific example, 100 nodes are removed from the network at each cycle by the `DynamicNetwork` class.

3.6 Observers

Observer objects will be run periodically (in a time-driven simulator, potentially at every time step) to analyze the network, the nodes composing the network and the state of the protocols executed on them. The aim is to collect statistical information about the behavior of the simulated system.

Some of the observers will be highly customized for the particular protocol to be monitored. For example, in our load-balancing prototype, an observer object reports the standard deviation of the loads measured at each of the nodes, and eventually stops the simulation when some configured level of balancing is obtained.

Other observers will be more general, and applicable to different function domains. For example, we developed a graph-theoretical library for analyzing interesting graph properties like diameter, clustering, conductance, etc.

4 Validation Plan

In order to validate the simulation results obtained through PEERSIM, we have started to implement some of our protocols using a realistic distributed planetary-scale open platform like

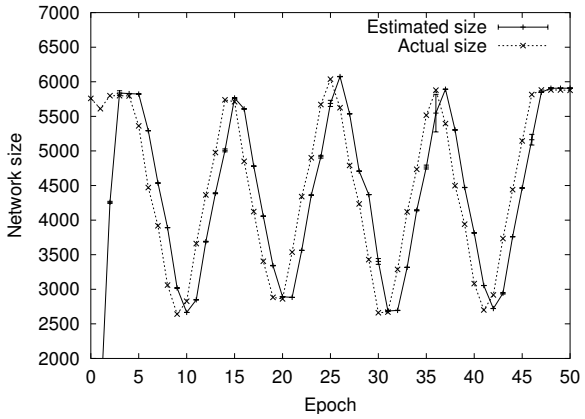

Figure 5: The estimated size (as provided by COUNT) and the actual size of a network oscillating between 2500 and 6000 nodes (approximately). Standard deviation of estimated size is displayed using vertical bars.

PLANET-LAB. PLANET-LAB is an international consortium aimed at providing a testbed for developing, deploying, and accessing planetary-scale services. At the time of writing, Planet-Lab is composed of more than 400 production machines, two of them are hosted in Bologna.

So far, we have implemented two protocols on Planet-Lab: aggregation [3, 5] and NEWSCAST [2]. Aggregation is the common name for a set of functions that provide global statistical information about a distributed system, like counting, averages, sums, extremal values, and so on. NEWSCAST is a membership management protocol aimed at maintaining a random connected topology among the set of all nodes. Newscast is used by aggregation as communication topology.

The preliminary results obtained in PlanetLab are encouraging and confirm the simulations performed through PEERSIMproduce realistic results. We have implemented the count protocol, whose task is to compute the current size of the network.

A summary of the experimental results obtained on PlanetLab is illustrated in Figure 5. During the experiment, 300 machines belonging to the PlanetLab testbed were used. Each machine was running up to 20 virtual nodes, each participating as a distinct entity. In other words, the maximum size of our emulated network was 6000 virtual nodes, distributed over five continents. The size of the network was made to oscillate between 2500 and 6000 nodes during the experiment. Virtual nodes were removed and added using a central scheduler that randomly picked nodes from the network to produce the oscillation effect shown in the figure.

The parameter used for this protocol were as follows: 20 concurrent protocol instances; cache size of NEWSCAST equal to 30; cycles of 5 seconds, while the number of cycles in an epoch is 30 (that is, the length of an epoch is approximately 2.5 minutes). For a detailed explanation of the protocol and the meaning of this parameters, please refer to [3, 5].

Several experiments were run, all of them starting at 02:00 Central European Time during workdays. All of them produced results similar to those shown in the figure. The communication mechanism of our implementation is based on UDP. This choice is motivated by the fact that

in a network based on NEWSCAST, interactions between nodes are short-lived, so establishing a TCP connection is relatively expensive. On the other hand, the protocol can tolerate message omissions. The observed message omission rate during our experiments varied between 3% and 8%.

The figure shows two curves, one representing the real size of the network at the beginning of a given epoch, and the other representing the estimated size, averaged over all nodes in the network. The (very small) standard deviation of the estimates over all nodes is also illustrated using vertical bars.

Based on the encouraging results we obtained, we started to implement a new framework, called FLU, designed to address the problem of deploying, running and managing a set of one or more epidemic protocols (such as Newscast or Aggregation) in a real world environment. The FLU platform allows the developer to write and test his/her protocols with its configurable fine-grained distributed logging facilities.

The framework is written in Java as the PeerSim simulator and they share some concepts and terminology (such as the cycle/epoch concepts and some API methods). Although the primary goal is to run FLU on Planet-lab, our framework is not limited to that environment.

The interface provided to the programmer is quite high level. Protocol developers do not have to deal with communication issues (a hidden datagram communication layer is provided by default) or any other low level details, but they can focus on what their protocol is supposed to do, basically extending a provided template class. As in Peersim, a configuration file to build the protocol stack is needed; in FLU, the configuration is expressed by an XML file. Once the overlay network is up, from any node it is possible to manage some particular node or the whole network. For example, having an epidemic broadcast service, it is possible to spread a new protocol class to every node or to spread an XML command saying something like "stop protocol i on the next cycle if your uptime is grater than 110 cycles". In this way, the facilities provided by the framework are used to manage the framework itself.

References

[1] IEEE 802.11 working group. ANSI/IEEE std. 802.11, 1999 edition: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. Technical report, ANSI/IEEE, 1999.

[2] Márk Jelasity, Wojtek Kowalczyk, and Maarten van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, November 2003.

[3] Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings of The 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 102–109, Tokyo, Japan, March 2004. IEEE Computer Society.

[4] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors, *Engineering Self-Organising*

Systems, number 2977 in Lecture Notes in Artificial Intelligence, pages 265–282. Springer, 2004.

[5] Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of The 2004 International Conference on Dependable Systems and Networks (DSN)*, pages 19–28, Florence, Italy, June 2004. IEEE Computer Society.

[6] Qualnet Home Page. `http://www.scalable-networks.com/`.

A PeerSim HOWTO: build a new protocol for the peersim simulation framework

This section can be found under `http://peersim.sf.net/`. Written by Gian Paolo Jesi.

A.1 Introduction

NOTE: This tutorial revision covers peersim release 0.3 topics.

This tutorial is aimed to give you a step by step guide to build from scratch a new peersim application (`http://sourceforge.net/projects/peersim`): a framework to experiment with large scale P2P overlay networks. In this tutorial it supposed that you and/or your workstation have:

- knowledge of O.O. programming and Java language;
- a working Java compiler ($\geq$ JDK 1.4.x);
- a working peersim source tree (you can download it from sourceforge CVS);
- the Java Expression Parser (download it from: `http://www.singularsys.com/jep/`);
- (suggested) gnuplot software.

The aim of this tutorial is to be as practical as possible; the goal is to give the reader the basics of peersim usage and the basics about how to write a simple component. This tutorial IS NOT exhaustive at all!
Because of the spiritus of this tutorial, after a brief introduction to basic concepts, we'll try to learn peersim and its basic components using a by example methodology.

A.2 Introduction to Peersim

A.2.1 Why peersim

One of the P2P system properties is that they can be extremely large scale (millions of nodes); another issue to deal with, is the high dynamicity of such systems: nodes in the network join and leave continously. Setting up a protocol experiments in a such simulated environment it's not an easy task at all.

Peersim has been developed to cope with these P2P properties and thus to reach extreme scalability and to support dynamicity. In addition, the simulator structure is based on components and makes easy to fast prototype a simulation joining together different pluggable building blocks. The term "components" used here has no relation with high level component architectures (e.g.: CORBA, DOM+).

The peersim performances can be reached only assuming some relaxing assumptions about the simulation details. For example, the overhead introduced by the low level communication

protocol stack (e.g.: TCP or UDP) in not taken into account because of the huge additional memory and CMU time requirements needed to accomplish this task. Another simplifying assumption is the absence of concurrency: in peersim the simulation is sequential and based on the concept of cycle in which every node can select a neighbor (the neighborhood relation could be defined by a fixed topology or defined by an overlay management protocol such as *Newscast*) and perform a protocol defined function.

A.2.2 Peersim simulation life cycle

The peersim structure is aimed to promote modular programming of building blocks. Every such block is easily replaceable by another component having a similar function, that means, in brief, having the same interface. In the peersim framework, a simulation is carried by the *Simulator* class. The general idea of the simulation model is:

1. choose a network size (number of nodes);
2. choose 1 or more protocol to experiment with and eventually initialize the protocol(s); this step will build a topology on top of raw nodes inserted at the previous point;
3. choose 1 or more *Observer* object to monitor what you are interested in;
4. optionally, choose 1 or more *Dynamics* object to modify during execution the parameters of the simulation (e.g.: the size of the network, update particular values inside protocols, ...);
5. ... run your simulation invoking the *Simulator* class

This is a very general model to give the reader an idea to start with, but it can be extremely more complex.

All the object created during the simulation are instances of classes that implements one or more well defined framework interfaces. The main interfaces I suggest you to become familiar with are in the Table1.

The life cycle of a peersim simulation is hard coded inside the *Simulator* class. It first reads a particular configuration file (see section A.2.3) containing all the simulation parameters concerning all the objects involved in the experiment. If no error occurs, the requested objects are created (all the nodes making the overlay connected with one or more protocol object, the *Observer* and *Dynamics* objects). From the developer point of view, it's important to note that the protocols creation process is based on **cloning**: only one instance of each protocol is actually forged (with the **new** statement) and then it's cloned to populate all the network. Thus the *clone()* method has to be designed with care to avoid unpredictable results.

The initialization phase is carried out by a special *Dynamics* object that runs only at the beginning. To obtain this effect, this initializer is internally wrapped by the simulator in a *Scheduler* class object that ensures a single shot. In the configuration file, the initialization *Dynamics* are easily recognizable by the `init` prefix. Please note that in the following pages we'll talk about *Initializer* objects just to remark their function and to distinguish them from ordinary *Dynamics* objects.

Node	All the elements of a P2P network are called nodes, the interface manages the local view of neighbor, the reference to the protocol, its index identifier inside the topology global array (invisible to protocols)
CDProtocol	A protocol simply defines an operation to be performed at each cycle (only method nextCycle() is defined)
Linkable	A class implementing this interface has access to the underling network: can access to its local view of neighbor
Observer	Objects running at each cycle collecting data about the current simulation state
Dynamics	Objects running at each cycle modifying values of other objects

Table 1: Peersim main classes or interfaces

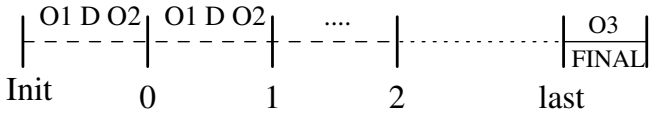

Figure 6: Observer and Dynamics scheduling

The way the simulator manages the interactions between the protocol(s) run, the *Observer* object(s) and the *Dynamics* object(s) in each cycle can be quite sophisticated. Each object in peersim (Dynamics, Observers and Protocols) is wrapped into *Scheduler* objects which adds fine grained scheduling facilities to each simulation component.

Before executing the protocol code, the simulator runs the *Dynamics* and *Observer* object(s), but the developer can choose and define the execution order of these components and the cycle interval to work in. For example, as depicted in Figure6, we can choose to run one or more *Observer* object before and/or one or more *Observer* objects after the *Dynamics* object(s). Nevertheless, also after the last cycle we can choose to run an *Observer* to retrieve a final snapshot.

In the Figure6, O1,2,3 are *Observer* components and D represents one or more *Dynamics* objects. Please note that before the first protocol run a first *Dynamics* and *Observer* run is performed (depicted in the interval between init and cycle 0).
The snapshots taken by the *Observer* objects are sent to standard output and can be easily redirected to a file to be collected for further work. When developing an Observer class, a good role is to print out data using the `peersim.util.Log` static class.

A.2.3 The config file

The config file is a plain ASCII text file, basically composed of key-value pairs; the lines starting with "#" character are ignored (comments). The pairs are collected by a standard Java *java.util.Properties* object when the simulator starts using for example the following command:

```
java -cp <class-path> peersim.Simulator config-file.txt
```

Clearly the classpath is mandatory only if you haven't set it yet in a global shell variable.

A.2.4 Configuration example 1

First of all, what we are going to do in this first experiment?

We are going to impose a fixed P2P random topology composed by 50000 node network; the chosen protocol is *Aggregation* (what is aggregation? see section A.7) using an average function. The values to be aggregated (averaged) at each node are initialized using a linear distribution on the interval [0, 100]. Finally an *Observer* monitors the averaging values. Looks easy!!

```
1  # PEERSIM EXAMPLE 1
2  # random.seed 1234567890
3  simulation.cycles 30
4  simulation.shuffle
5
6  overlay.size 50000
7  overlay.maxsize 100000
8
9  protocol.0 peersim.core.IdleProtocol
10 protocol.0.degree 20
11
12 protocol.1 example.aggregation.AverageFunction
13 protocol.1.linkable 0
14
15 init.0 peersim.init.WireRegularRandom
16 init.0.protocol 0
17 init.0.degree 20
18
19 init.1 example.loadbalance.LinearDistributionInitializer
20 init.1.protocol 1
21 init.1.max 100
22 init.1.min 1
23
24 observer.0 example.aggregation.AverageObserver
25 observer.0.protocol 1
```

Lets comment the code line by line. The first thing to node are the key names: some of them are indexed and some other not (e.g. protocol.0.xxx versus simulation.<parameter>). That means the unindexed keys refers to static simulation elements, in fact the simulation itself is one and the same holds for the P2P network: only one network!
Please note that from Peersim release 0.2 the component indexes in the configuration can be regular strings.

For the other simulation components you can think about the existence of a dedicated array for each of their type (one for protocols, initializer, ...); the index is the only reference to deal with them. So the key for indexed components can be (informally) expressed as:

```
<init | protocol | observer | dynamics> . index_number [. <parameter_name> ]
```

The final <parameter_name> is contained between [] to express that it's optional. This is the case when the element is declared. For example, at **line 9**, the first protocol chosen comes to life; the **key part** contains its type (or interface type) followed by the index (always starting from 0, as in arrays) and **the value** part contains the desired component class full package path (you have to check the javadoc files or the source tree to discover the correct package path). In the case of a component parameter declaration, the **key part** contains the parameter name and the **value part** is simply the value desired (usually an integer or a float).

At this point, should be clear that **from line 3 to line 7** some global simulation properties are imposed; these are the total number of simulation cycles and the overlay network size. The parameter `simulation.shuffle` (**line 4**) it is a little different from what we have stated until now; it is used as a flag, so it does not need a parameter. Its job is to shuffle the order in which the nodes are visited in each cycle. The parameter `overlay.maxsize` (**line7**) sets an upper bound on the network size, but in this example it is useless (you can comment it out) and it's only present for sake of completeness (will be useful next).

From **line 9 to line 13**, two protocols are put in the arena. The first one, peersim.core.IdleProtocol does nothing. It is useful because of its ability to access to the topology, in fact it provides neighbor links to each node. This feature is present because *IdleProtocol* is an implementation of the *Linkable* interface. Next line declares the graph degree.

The second protocol (index 1: `protocol.1 aggregation.AverageFunction`) is the averaging version of aggregation. Its parameter (linkable) is extremely important: it expresses the need to access the topology using not this protocol itself (aggregation). This is due to the structure of aggregation: it does not implement the *Linkable* interface, so it can't see the neighbor list by itself and it must use some other protocol to do that. The value of parameter linkable is the index of a *Linkable* interface implementing protocol (*IdleProtocol* in the example). Clearly to know if a protocol can get access to the topology directly, you have to check the documentation (or source code).

From **line 15 to line 22**, it's time to initialize all the components previously declared. Again, the initialization components are 2 and are indexed as usual. The first initializer `peersim.init.WireRegularRandom`, imposes a topology. The nodes using the declared protocol are linked randomly to each other to form a random graph having the specified degree parameter. Please note that this degree declaration is exactly the same of the previous (the one dedicated to the first protocol creation).

The second initializer task is to initialize the aggregation function value-field to be averaged. The initialization values follows a linear distribution fashion. The parameters declared are three: `protocol`, `max`, `min`. Respectively, their meaning is:

- a protocol to point to: the initializer needs a reference (index) to a protocol extending *aggregation.AbstractFunction* Class to get access to the value to be aggregated (averaged); it is clear that this protocol must be *aggregation.AverageFunction* (index 1);

- the maximum value in the linear distribution;
- the minimum value in the linear distribution

Finally at **line 24,25** the last component is declared: `aggregation.average Observer`. Its only parameter used is protocol and clearly refers to an *aggregation. AverageFunction* protocol type, so the parameter value is index 1 (in fact: `protocol.1 aggregation.AverageFunction`).

Now you can try the example writing on a console the following line:

```
java -cp <class-path> peersim.Simulator example1.txt
```

The classpath is mandatory only if the used system has not peersim classes in the shell CLASSPATH environment variable. To get the exact output that will follow, the reader should uncomment the parameter at **line 2**:

```
random.seed 1234567890
```

on top of the configuration file. This parameter is very useful to replicate exactly the experiments results based on (pseudo) random behavior. The experiment output is (some initialization string may be different):

```
Simulator: loading configuration
ConfigProperties: File example/config-example1.txt loaded.
Simulator: starting experiment 0
Simulator: resetting overlay network
Network: no node defined, using GeneralNode
Simulator: running initializers
- Running initializer 0: class peersim.init.WireRegularRandom
- Running initializer 1: class example.loadbalance.LinearDistributionInitializer
Simulator: loaded observers [observer.0]
Simulator: loaded modifiers []
Simulator: starting simulation
observer.0 0 28.57969570575493 1.0 50.49999999999998 100.0 1.0 50000 50000
Simulator: cycle 0 done
observer.0 1 15.744375112466432 0.5508937280006126 50.500000000000185 99.64260285205704 1.993979879597592 50000 50000
Simulator: cycle 1 done
observer.0 2 8.77307045667709 0.3069686446980087 50.50000000000009 86.06868887377748 11.048700974019479 50000 50000
Simulator: cycle 2 done
observer.0 3 4.909681896225926 0.17178915922597776 50.499999999999794 74.03587220181905 22.769780085543115 50000 50000
Simulator: cycle 3 done
observer.0 4 2.7403309556342257 0.09588383948687113 50.500000000000426 65.43171163227953 33.427798365537626 50000 50000
Simulator: cycle 4 done
observer.0 5 1.538286672869342 0.053824459459153234 50.49999999999973 59.82515640226745 42.62594413722992 50000 50000
Simulator: cycle 5 done
observer.0 6 0.866397905938638 0.03031515502679675 50.50000000000007 55.26130498088358 45.94325388089578 50000 50000
Simulator: cycle 6 done
observer.0 7 0.485544546348093 0.016989143318636584 50.4999999999996 53.34350686753126 47.92146780934889 50000 50000
Simulator: cycle 7 done
observer.0 8 0.27325943590085566 0.009561313693267594 50.499999999999936 51.953084686348944 49.100818456230826 50000 50000
Simulator: cycle 8 done
observer.0 9 0.15407802503043988 0.005391170942362905 50.499999999999545 51.464657035213264 49.43879802069546 50000 50000
Simulator: cycle 9 done
observer.0 10 0.08620333588583261 0.0030162440067013846 50.500000000000156 51.099961126584006 49.98131655222747 50000 50000
Simulator: cycle 10 done
observer.0 11 0.04848730705794467 0.0016965648464962858 50.4999999999997 50.816956855036466 50.22577832539035 50000 50000
Simulator: cycle 11 done
observer.0 12 0.027214744249562235 9.522405182250473E-4 50.499999999999524 50.65301253758219 50.29955826845794 50000 50000
Simulator: cycle 12 done
observer.0 13 0.015246845383671713 5.334852246380476E-4 50.50000000000032 50.59479421528527 50.38736504947625 50000 50000
Simulator: cycle 13 done
observer.0 14 0.008587160488627146 3.004636780264248E-4 50.499999999999815 50.543660258997136 50.44122418780829 50000 50000
Simulator: cycle 14 done
observer.0 15 0.004850437249792671 1.697161964119851E-4 50.50000000000037 50.52544970665122 50.46753516145482 50000 50000
Simulator: cycle 15 done
```

```
observer.0 16 0.0027428141606463717 9.59707265215587E-5 50.50000000000047 50.515509548126744 50.48203242786418 50000 50000
Simulator: cycle 16 done
observer.0 17 0.001550607390364058 5.4255559832703925E-5 50.4999999999997 50.50982384430303 50.49003444731606 50000 50000
Simulator: cycle 17 done
observer.0 18 8.746858998689715E-4 3.0605150904137896E-5 50.5000000000003 50.50564226243819 50.495105203016905 50000 50000
Simulator: cycle 18 done
```

The observer component produces many numbers, but looking at the 6th and 7th data columns (respectively the maximum of averages and the minimum of averages) it's easy to see how the variance decreases very quickly. At circle 12 (look at the underlined data), quite all the nodes has a very good approximation of the real average (50). Try to experiment with different numbers and then to change the init distribution (e.g.: using `aggregation.PeakDistribution Initializer`) and / or the protocol stack (put *Newscast* or *SCAMP* instead of *IdleProtocol*).

A.2.5 Configuration example 2

This second example is an improved version of the first one. What's new? Now the aggregation protocol runs on top of Newscast and it's easy to switch to the peak distribution (comment 4 lines and uncomment 2 lines). Moreover, there is a Dynamics object that changes the network size (it shrinks it by cutting out 500 nodes each time).

```
1  simulation.cycles 30
2  simulation.shuffle
3
4  overlay.size 50000
5  overlay.maxsize 200000
6
7  protocol.0 example.newscast.SimpleNewscast
8  protocol.0.cache 20
9
10 protocol.1 example.aggregation.AverageFunction
11 protocol.1.linkable 0
12
13 init.0 peersim.init.WireRegularRandom
14 init.0.protocol 0
15 init.0.degree 20
16
17 #init.1 example.aggregation.PeakDistributionInitializer
18 #init.1.value 1
19 init.1 example.loadbalance.LinearDistributionInitializer
20 init.1.protocol 1
21 init.1.max 100
22 init.1.min 1
23
24 observer.0 example.aggregation.AverageObserver
25 observer.0.protocol 1
26
27 dynamics.0 peersim.dynamics.GrowingNetwork
28 dynamics.0.add -500
29 dynamics.0.minsize 4000
30 dynamics.0.from 5
31 dynamics.0.until 10
```

The global parameters are the same as in the previous example; only new additions are discussed below. At **line 7-8** there is the *Newscast* (what is newscast? see section A.8) component declaration with its only parameter cache (please note: cache size should be at least as large as

network degree size). At **line 17-18** there is a different distribution type: `aggregation.Peak DistributionInitializer`, but it's inactive. To switch it on, simply delete the preceeding symbol "#" and comment out the following 4 lines. The peak distribution initializes all nodes except one with 0 value and the node left takes the value declared in parameter value.

From **line 27 to 32** is present the last new component: `dynamics.0 peersim.dynamics.GrowingNetwork`. As stated previously, a *Dynamics* interface implementing object is able to change some other object properties; the change can be performed at each simulation cycle (default behavior) or using a more sophisticated idea. The object chosen in the example deletes 500 nodes from the net at each time (well, it is not completely correct to talk about deletion in the peersim vision, in fact the *Linkable* interface does not support node deletion from the overlay; so it's better to think about "unlinking" nodes from the overlay). The parameters `add`, `minsize`, `from` and `until` have respectively the following meaning:

- adds the specified number of nodes (if negative subtracts);
- the minimum size ir referred to the overlay: it can't be less than what's stated here;
- the cycle number from which the *Dynamics* object can start running;
- the cycle number until which the *Dynamics* object can run.

Other parameters are available, please check the source or the JavaDoc.

It's interesting to note that not all the parameters associated to a *Dynamics* component can be found in the *Dynamics* itself source code (or documentation); this is due to the Simulator class behavior. When it creates the *Dynamics* object instances (this hold also for *Initializer* and *Observer* objects), it wraps them in a *Scheduler* class object: this is the class where some parameters (such as `step`, `from`, `until`) are actually defined.

A.3 Advanced configuration features

Thanks to the presence of the Java Expression Parser (since release 0.2), the configuration file can handle many types of **expressions**, such as boolean expressions, common mathematical functions and well known predefined constants (e.g.: π and e); for an exhaustive feature list check the parser web page (`http://www.singularsys.com/jep/index.html`).
Expressions can be used anywhere instead of numeric values, as follows:

```
MAG 2
SIZE 2^MAG
```

the variable SIZE will be automagically evaluated in number 4.
Multiple expressions can be written in a tree-like fashion and they'll be evaluated recursively (the CMU conscious users have to know that no optimizations are performed and the same expression may be evaluated many times) as in the following code sample:

```
A B+C
```

```
B D+E
C E+F
D 1
E F
F 2
```

The evaluation will produce: A=7, B=3, C=4, D=1, E=2 and F=2.

Recursive definitions are not allowed and a simple trick is used to avoid them: if the recursion depth is grater than a configurable threshold parameter (set at 100 by default) an error message is printed and the simulator stops.

For any kind of simulator object (means Dynamics, Observer, Protocol), it is possible to specify an ordering scheme and to specify an arbitrary name to multiple instances of a given entity. The object prefixes are not limited to numerical indexes, but they can be expressed by any string, as follows:

```
observer.conn ConnectivityObserver
observer.0 Class1
observer.1 Class2
```

The order is lexicographically, but can be explicitly rearranged giving an object name list separated by any non-word character (non alphanumeric or underscore):

```
order.observer 2,conn,0
```

If not all names appear in the list, then the vacant objects will follow the default alphabetical order. For example:

```
order.observer 2
```

will produce the following order:

$$\langle \texttt{observer.2} ; \texttt{observer.0} ; \texttt{observer.conn} \rangle$$

Another available feature is the chance to exclude items from the list. To obtain this effect type:

```
include.observer conn 2
```

This will return `observer.conn` and `observer.2` in this exact order. If the list is empty, then an empty ordering array will be generated; means that, in this case, no observer will run. Note that `include` is stronger than `order` in fact if the former is defined, then the latter is ignored.

A.3.1 A concrete example

To have a practical idea about how to use these new features, the following example is presented; it is a modified example2 version.

```
1 #random.seed 1234567890
2 simulation.cycles 30
3 simulation.shuffle
4
5 # Imposes the correct protocol running order:
6 order.protocol ncast,avgagr
7
8 overlay.size 50000
9 overlay.maxsize 200000
10
11 protocol.ncast example.newscast.SimpleNewscast
12 protocol.ncast.cache 20
13
14 protocol.avgagr example.aggregation.AverageFunction
15 protocol.avgagr.linkable ncast
16
17 init.wrr peersim.dynamics.WireRegularRandom
18 init.wrr.protocol ncast
19 init.wrr.degree 20
20
21 # UNCOMMENT THE FOLLOWING LINES TO GET A PEAK DISTRIBUTION
22 #init.pkdistrib example.aggregation.PeakDistributionInitializer
23 #init.pkdistrib.value 10000
24 #init.pkdistrib.protocol avgagr
25
26 # UNCOMMENT THE FOLLOWING LINES TO GET A LINEAR DISTRIBUTION
27 init.ldistrib example.loadbalance.LinearDistributionInitializer
28 init.ldistrib.protocol avgagr
29 init.ldistrib.max 100
30 init.ldistrib.min 1
31
32 observer.avgobs example.aggregation.AverageObserver
33 observer.avgobs.protocol avgagr
34
35 dynamics.grnetwork peersim.dynamics.GrowingNetwork
36 dynamics.grnetwork.add -500
37 dynamics.grnetwork.minsize 4000
38 dynamics.grnetwork.from 5
39 dynamics.grnetwork.until 10
```

In this configuration file, the protocol indexes are no more used; but the same holds for each kind of object. A symbol string that summarize the original class name is used instead of an index number. Because of the chosen protocol symbol names (`ncast` and `avgagr`), it is necessary to impose a different running order scheme to let newscast run first using (at **line 6**):

```
order.protocol ncast,avgagr
```

A.4 Writing a new protocol

This section covers the description of how to write a new protocol.

A.4.1 Which kind of protocol?

The protocol we are going to develop is a simple load balancing algorithm. It works as follows. The state of a node is composed of two values: the local load and the quota; the second one is the amount of "load" the node is allowed to transfer at each cycle. The quota is necessary in order to make real load balancing, otherwise it would be simply averaging. Every node contacts the most **distant** neighbor in its local view and then exchanges at maximum the quota value. The concept of "distance" is expressed in terms of maximally different load from the current node local load. Comparing the distance to the actual node load, the protocol chooses to perform a load balance step using a push or pull approach.

After each cycle, the quota value is restored to allow further computation. The protocol does not care about topology management and relies on other components to get access to this feature (e.g.: *Newscast*, *IdleProtocol*).

A.4.2 Needed components

Now we have a general idea on what we want to code and it's time to adapt it to the peersim framework. Writing a the protocol class itself, it is usually not sufficient. Some companion components are required. For example, to restore the quota value for each node at the end of each cycle, a specific *Dynamics* object is required. Peersim is basically a collection of interchangeable components, so the development of new stuff should have **modularity** in mind and should maximize code reuse. To achieve this, the following classes are proposed:

- **protocol class itself**: it is built on *peersim.vector.SimpleValueHolder*; it's a simple base class to access a single float variable. It shares the same interface as aggregation: many other components can be used together with the load balancing protocol, such as the initializers classes.
- **Dynamics component**: it is necessary to restore the quota value at each node at the end of each cycle (as previously stated). This object is quite straightforward: it simply implements the only one method the interface *Dynamics* declares, invoking the protected protocol method *resetQuota()*.
- **Initializer and Observer components**: they are not really needed! In fact the aggregation initializers can be used directly because they share the same interface (both extends *SingleValueHolder*). Also the aggregation observers can be used (the *aggregation.AverageObserver* in particular). In addition a new observer object can be written to monitor the **quota** parameter and thus the amount of traffic exchanged in the overlay. Please note that the initializers provided in the example package are "light", demo versions; the developer is encouraged to use the *peersim.vector* package initializers.

To give the reader an idea about the actual code to write, the following subsections are presented, in which the author put comments and explanations in the way of the class code itself.

```
package example.loadbalance;
```

```
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.*;
import peersim.vector.SingleValueHolder;
import peersim.cdsim.CDProtocol;

public class BasicBalance extends SingleValueHolder implements CDProtocol{
// Fields:
public static final String PAR_QUOTA = "quota"; // allowed config file parameter
private final double quota_value; // original quota value taken from configuration

protected double quota; // current cycle quota

// Constructor:
public BasicBalance(String prefix, Object obj) {
 super(prefix);
  // get quota value from the config file. Default 1.
 quota_value = (double)(Configuration.getInt(prefix+"."+PAR_QUOTA, 1));
 quota = quota_value;
}
```

It's simply standard Java code until now; the class needs also to implement *CDProtocol* (and *Protocol*) interface(s) and to provide the *nextCycle()* method that is where the actual protocol algorithm is located.

In the constructor signature, two parameters are present; the first one is a string corresponding to the configuration file protocol key (e.g.: `protocol.1` in the *LoadBalance* protocol case), the second one is the own protocol id index casted in a *Object* type.

```
// Resets the quota.
protected void resetQuota() {
    this.quota = quota_value;
}
```

The *resetQuota()* method is called by the dynamics object at the cycle end. Clearly a suitable dynamics entry should be present in the configuration file (such as: `dynamics.0 loadbalance.ResetQuota` and `dynamics.0. protocol protocol-index`). This method is not mandatory, but it's much more software engineering oriented then a dirty variable access performed by the dynamics object.

```
public Object clone() throws CloneNotSupportedException {
  BasicBalance af = (BasicBalance)super.clone();
  return af;
 }

 // Implements CDProtocol interface
 public void nextCycle( Node node, int protocolID ) {
  int linkableID = FastConfig.getLinkable(protocolID);
  Linkable linkable = (Linkable) node.getProtocol( linkableID );
if (this.quota == 0) {
  return; // skip this node
  }
  // this takes the most distant neighbor based on local load
  BasicBalance neighbor = null;
  double maxdiff = 0;
  for(int i = 0; i < linkable.degree() ; ++i)
  {
```

```
  Node peer = linkable.getNeighbor(i);
  // The selected peer could be inactive
  if(!peer.isUp()) continue;
  BasicBalance n = (BasicBalance)peer.getProtocol(protocolID);
  if(n.quota!=1.0) continue;
  double d = Math.abs(value-n.value);
  if( d > maxdiff ) {
neighbor = n;
maxdiff = d;
}
}
  if( neighbor == null ) {
  return;
 }
 doTransfer(neighbor);
 }
```

The first method is required by the *Protocol* interface and basically calls the ancestor cloning method. So, nothing special here.

The second one is required by *CDProtocol* interface. It is the behavior performed by the protocol. The arguments represents a reference to the node itself (the node on which the simulator is invoking the *nextCycle()* method) and the index protocol identifier (the BasicBalance protocol id in this case). First it has to get a reference (in indexed form) to the *Linkable* interface enabled protocol in the node protocol stack; as a remind, something implementing the *Linkable* interface, is an entity capable of accessing the topology. Having this linkable reference we can access to the real *Linkable* interface implementation with:

```
int linkableID = FastConfig.getLinkable(protocolID);
Linkable linkable = (Linkable)node.getProtocol(linkableID);
```

Using the static *FastConfig* class we can get the current protocol corresponding Linkable identifier; this class manages the protocol `linkable` parameter without direct user intervention. Then we can access the actual Linkable object as shown in the second line.

If the local quota is equal to 0, means that the node we have already spent its amount of network traffic, so it returns.

To get the most distant node from the current one, a for loops on all neighbor node load value; the number of neighbor is equal to the node degree (accessible thanks to *Linkable* interface). To pick a node having a the *Linkable* access:

```
Node peer = linkable.getNeighbor(i);
```

and from this obtained *Node* interface reference it is possible to get the protocol interface we are interested in (*BasicBalance*):

```
BasicBalance n = (BasicBalance)peer.getProtocol(protocolID);
```

When the protocol finds a suitable neighbor, it performs a load balancing step invoking the *doTransfer()* method.

```
// Performs the actual load exchange selecting to make a PUSH or PULL approach.
// It affects the involved nodes quota.
protected void doTransfer(BasicBalance neighbor) {
    double a1 = this.value;
    double a2 = neighbor.value;
    double maxTrans = Math.abs((a1-a2)/2);
    double trans = Math.min(maxTrans,quota);
    trans = Math.min(trans, neighbor.quota);

    if( a1 <= a2 ) // PULL
    {
        a1+=trans;
        a2-=trans;
    }
    else // PUSH
    {
        a1-=trans;
        a2+=trans;
    }

    this.value = a1;
    this.quota -= trans;
    neighbor.value = a2;
    neighbor.quota -= trans;
}
```

The last method takes as parameter a reference to the picked neighbor. This is the place where it's time to decide to perform a pull or a push load balancing approach. To make this choice the local load value is compared with the neighbor load value. In case of a push choice, the local value is increased and the other node value is decreased; in the other case (pull) the exact opposite holds. The *maxTrans* variable is the absolute amount of "load" to transfer to reach the balance between the two involved nodes; because of the quota upper bound on the transfers at each cycle, the algorithm chooses the minimum between the quota itself and the aimed *maxTrans* amount. The quota value is decreased by the same amount at both nodes.

A.4.3 Load balancing dynamics class code

```
package loadbalance;

import peersim.config.*;
import peersim.core.*;
import peersim.dynamics.Dynamics;

public class ResetQuota implements Dynamics {
// Fields:
public static final String PAR_VALUE = "value";

public static final String PAR_PROT = "protocol";

private final double value;
private final int protocolID;

// Constructor:
public ResetQuota(String prefix)
{
    value = Configuration.getDouble(prefix+"."+PAR_VALUE);
    protocolID = Configuration.getPid(prefix+"."+PAR_PROT);
}
```

```
// Dynamics interface method:
public void modify() {
    for(int i=0; i < Network.size(); ++i)
    {
        ((BasicBalance)Network.get(i).getProtocol(protocolID)).resetQuota();
    }
  }
}
```

The code is very compact because the *Dynamics* interface itself is very simple: only the *modify()* method. The constructor takes care of initializing the configuration file parameters (respectively: the reset value and the protocol identifier to deal with). The *modify()* method makes use of network global knowledge: it invokes the *resetQuota()* method on all the *Network* object elements (it's a static object available everywhere in the simulator environment; you can think about it as an array). It is clear that the simulator has global knowledge, but it is up to the protocol developer to make use or not of this facility according to the consistency of the simulation itself.

A.4.4 Implementing the Linkable interface

In this HOWTO there are a lot of references about the *Linkable* interface and about its importance, so for the sake of completeness, it's time to give a look at how to implement it in brief. It's interesting to node that this interface should be implemented by low level or by topology management protocols and not by a higher level protocol such as a load balancing one. The reason to discourage the implementation is the risk to affect modularity. At least, the reader should consider the ability to switch off the built in *Linkable* interface and to use an external protocol facility instead.

The interface defines five methods: *degree()*, *getNeighbor()*, *addNeighbor()*, *contains()*, *pack()*. These methods are not usually invoked by the protocol itself (except for *getNeighbor()*), but by an *Initializer* object instead (such as: *peersim.dynamics.WireRegularRandom*). Please note that there is no way to remove nodes from the overlay; the only chance to get a similar effect, is to disable a peer accessing to the *peersim.core.Fallible* interface (extended by the *Node* interface) and setting one of the available node states (*peersim.core.Fallible.OK — DEAD — MALICIOUS — DOWN*).

A feasible implementation could be the following. First of all, the class (e.g.: *BasicBalance*) needs a structure to represent the neighbor view: an *ArrayList* structure it's fine.

```
protected ArrayList nView = null;
// Constructor:

public BasicBalance(String prefix, Object obj) {
    ...
    nView = new ArrayList();
    ...
}
// Linkable interface implementation methods:
public int degree() {
    return nView.size();
}

public Node getNeighbor(int i) {
```

```
    return (Node)nView.get(i);
}

public boolean addNeighbor(Node n) {
    if (!contains(n)) {
        nView.add(n);
        return true;
    }
    else {return false;}
}

public boolean contains(Node n) {
    return nView.contains(n);
}

public void pack() { ; } // unused!
```

Again the code is quite straightforward. All the elements inside the view are *Node* class (interface) types. All methods are simple functions built upon the *ArrayList* structure. The last method is included in the interface description with the aim to provide a view size compression facility, but it's usually not implemented (the size of each view is typically quite small).

A.5 A second new protocol

This new protocol is an extensions of the previous one. The general core is quite the same, but the algorithm uses the global load average value instead of the most distant neighbor load value. To calculate the global load average, a little trick is used; it would be possible to calculate this value using aggregation, but we can **simulate** the aggregation effect (alias calculating the average load) by running a static method with global knowledge once. This method will initialize a global variable available to all nodes.

This protocol is targeted to gain advantage from the newscast protocol features; when a node reaches the global load value (average), it switches to a DOWN state. In this way, the node exits from the overlay and the newscast protocol no more cares about it. The effect is that the topology shrinks as soon as the nodes reach the average load.

```
package loadbalance;

import peersim.util.CommonRandom;
import aggregation.AbstractFunction;
import peersim.core.*;

public class AvgBalance extends BasicBalance {

    public static double average = 0.0;
    public static boolean avg_done = false;

    // Costructor:
    public AvgBalance(String prefix, Object obj) {
        super(prefix, obj);
    }

       // Method to simulate average function aggregation
    private static void calculateAVG(int protocolID) {
        int len = Network.size();
```

```
        double sum = 0.0;
        for (int i = 0; i < len; i++) {
            AvgBalance protocol = (AvgBalance)Network.get(i).getProtocol(protocolID);
            double value = protocol.getValue();
            sum += value;
        }
        average = sum / len;
        avg_done = true;
    }
```

The first part is straightforward. Two global variables are defined: average, and avg_done; the second is a flag used to be sure not to perform the calculation more than once. A different and much more elegant approach is to define the average calculation method inside a static constructor, but this solution is **wrong**! When the node protocol objects are created, the load distribution is not defined yet, so the global average result will be 0.

The function *calculateAVG()* simulates the average aggregation behavior. It makes use of global knowledge, looping on each overlay node.

```
protected static void suspend( Node node ) {
    node.setFailState(Fallible.DOWN); }
}
```

This is the utility function to exit from the topology; simply sets a node state from *Fallible* interface.

```
// CDProtocol Interface implementation. Overrides the BasicBalance implementation:
    public void nextCycle( Node node, int protocolID ) {
        // Do that only once
        if (avg_done == false) {
            calculateAVG(protocolID);
            System.out.println("AVG only once "+average);
        }

        if( Math.abs(value-average) < 1 ) {
            AvgBalance.suspend(node); // switch off node
            return;
           }

        if (quota == 0 ) return;

        Node n = null;
        if (value < average ) {
            n = getOverloadedPeer(node, protocolID);
            if (n != null) { doTransfer((AvgBalance)n.getProtocol(protocolID)); }
        }
        else {
            n = getUnderloadedPeer(node, protocolID);
            if (n != null) { doTransfer((AvgBalance)n.getProtocol(protocolID)); }
        }

        if( Math.abs(value-average) < 1 ) AvgBalance.suspend(node);
        if (n != null) {
            if( Math.abs(((AvgBalance)n.getProtocol(protocolID)).value-average) < 1 )
           AvgBalance.suspend(n);
        }
    }
```

Method *nextCycle()* is the protocol algorithm core. It first checks for the average calculation: if the flag is not set, it performs the computation.

If the difference between the current and the average load is less then 1 (the fixed quota value per cycle) the node is suspended and thus exits from the topology defined by the newscast protocol; moreover, if the quota has been already spent, it returns. The protocol then checks if the local value is less or grater then the average and respectively get the most loaded or the least loaded neighbor and exchange.

```
 private Node getOverloadedPeer(Node node, int protocolID) {
         int linkableID = FastConfig.getLinkable(protocolID);
         Linkable linkable = (Linkable) node.getProtocol( linkableID );

         AvgBalance neighbor=null;
         Node neighborNode = null;
         double maxdiff = 0.0;
         for(int i = 0; i < linkable.degree(); ++i) {
             Node peer = linkable.getNeighbor(i);
             if(!peer.isUp()) continue;
             AvgBalance n = (AvgBalance)peer.getProtocol(protocolID);
             if(n.quota==0) continue;
             if(value >= average && n.value >= average) continue;
             if(value <= average && n.value <= average) continue;
             double d = Math.abs(value-n.value);
             if( d > maxdiff ) {
                 neighbor = n;
                 neighborNode = peer;
                 maxdiff = d;
             }
         }
         return neighborNode;
     }

     private Node getUnderloadedPeer(Node node, int protocolID) {
        int linkableID = FastConfig.getLinkable(protocolID);
Linkable linkable = (Linkable) node.getProtocol( linkableID );

         AvgBalance neighbor=null;
         Node neighborNode = null;
         double maxdiff = 0.0;
         for(int i = 0; i < linkable.degree(); ++i) {
             Node peer = linkable.getNeighbor(i);
             if(!peer.isUp()) continue;
             AvgBalance n = (AvgBalance)peer.getProtocol(protocolID);
             if(n.quota==0) continue;
             if(value >= average && n.value >= average) continue;
             if(value <= average && n.value <= average) continue;
             double d = Math.abs(value-n.value);
             if( d < maxdiff ) {
                 neighbor = n;
                 neighborNode = peer;
                 maxdiff = d;
             }
         }
         return neighborNode;
     }
 } // end of class
```

The methods to get the the most loaded or the least loaded neighbor are straightforward and very similar, but are shown for completeness.

A.6 Evaluating the protocols

The performance about load variance reduction can be analyzed with an *aggregation.Average Observer* or a *loadbalance.LBObserver* (they are very similar), but don't expect huge differences. In fact, from this point of view, the two protocols have nearly an identical performance, no matter whatever distribution you are using. The *AVGBalance* protocol improvement over the *BasicBalance* one is about the achieved overall load transfer. The *AVGBalance* amount of transfer is minimal and it is practically the same of the theoretical minimal amount of transfer needed to solve the problem; more on this topic can be retrieved at the following url:

`http://www.cs.unibo.it/bison/publications/modular-p2p.pdf`.

The *Observer* class code to inspect the load transfer amount is the following.

```
package loadbalance;

import peersim.core.*;
import peersim.reports.*;
import peersim.config.*;
import peersim.util.IncrementalStats;

public class QuotaObserver implements Observer {

public static final String PAR_PROT = "protocol";
private final String name;
private final int pid;   // protocol id to monitor
private IncrementalStats stats;   // object to compute statistics

// Constructor:
public QuotaObserver(String name) {
    this.name = name;
    pid = Configuration.getPid(name + "." + PAR_PROT);
    stats = new IncrementalStats();
}

// Observer interface implementation:
public boolean analyze() {
    for (int i = 0; i < Network.size(); i++) {
        BasicBalance protocol =  (BasicBalance) Network.get(i).getProtocol(pid);
        stats.add( protocol.quota );
    }
    System.out.println(name+": "+stats);
    return false;
}

} // end of class
```

The idea is very simple: at each simulation cycle it collects all node values about quota and print statistics on the console.

A.7 A few words about Aggregation

It is a very fast epidemic-style protocol targeted to compute a particular function (e.g.: average, max, min, ...) on a numeric value held at each network node. In order to work, every node

needs access to its neighbor list view on the overlay network; no particular requirements about the topology management protocol are imposed. In the case of averaging function, a generic method *updateState(a, b)* returns (a+b)/2, where a and b are values at (respectively) node a and node b. This kind of computation is performed by each node at each simulation cycle.

The global average is not affected, but the variance over all the estimates decreases very fast in a few cycles (the convergence rate is exponential and it does not care about the network size). The aggregation protocol is also very robust in case of node failures.

Suggested readings: project BISON publication page `http://www.cs.unibo.it/bison/pub.shtml`.

A.8 A few words about Newscast

Newscast is an epidemic content distribution and topology management protocol. Every peer in the system has a partial view knowledge about the topology which is modeled as a fixed size (c) set of node descriptors. Each descriptor is a tuple consisting of a peer address and a timestamp recording the time when the descriptor was created.

Each node updates its state by choosing a random neighbor and exchanging with it the view. The exchanging process merges the two involved peer partial view, keeping the c freshest descriptors. in this manner, old information (descriptor) are automagically removed from the system as time goes on. This process allows the protocol to repair the overlay topology removing dead links with minimum effort and this is a great feature for a highly dynamic oriented system where nodes join and leave continously.

The protocol relies on the timestamps, but it doesn't need synchronized clocks: timestamps have to be only mutually consistent. To achieve this, a simple time normalization of the received descriptors is performed. So time precision is not critical at all.

The emergent topology from newscast topology management has a very low diameter and it is very close to a random graph with (out) degree c. Suggested readings: "Large-Scale Newscast Computing on the Internet" `http://citeseer.nj.nec.com/jelasity02largescale.html`.

B PeerSim HOWTO 2: build a topology generator

This section can be found under `http://peersim.sf.net/`. Written by Gian Paolo Jesi.

B.1 Introduction

This tutorial teaches you how to build from scratch a new peersim (peersim project page: http://sourceforge.net/projects/peersim) topology generator. In order to understand this tutorial, the reader is encouraged to start reading the first peersim tutorial [1] to have an idea of the basic concepts that will not be discussed any further in this document.

The aim of this tutorial is to be as practical as possible; the goal is to give the reader ideas about technical or intermediate level features of peersim and to encourage him/her to experiment further. The full source code discussed in this document is available via CVS at peersim project page in the *peersim.example.hot* class package.

B.2 What is a topology?

The network abstraction in peersim is a (sometimes huge) array of *Node* structures (interfaces); because of the size of the network and to overcome scalability problems, usually in large P2P networks each node knows about the existence of a very small subset of other nodes (ex: order of log(N) where N is the whole network size). Thus each node has a short list of other node references, usually called "neighbors", build accordingly to some kind of strategy or rule.

Thus, we can say that a topology is how nodes are arranged (linked) together and clearly this depends upon the particular chosen rule. Examples of topology are the following (not exhaustive at all):

- random graphs
- Watts-Strogatz model graph
- star model
- ring model
- lattice model
- ...

B.2.1 Which rule to choose?

In this document, we have chosen to code a particular topology generator to build internet-like tree topologies. The building process is based on the *preferential attachment* approach. The rule applied is quite simple and takes into account geometric and network constraints to better

[1] `http://peersim.sourceforge.net/peersim_HOWTO.html`

mimic real world network. The preferential attachment choice can be affected by a parameter (α) that amplifies or reduces the geometric location influence in favor of the path distance.

The rule strategy is the following: we consider a square unit region D, then we start with node $x(0)$ chosen at random and we set $W(x(0)) = 0$ (it is the root node). For each i with $i = 1...n-1$ we choose a new node x(i) in the region D and we connect it to an **early inserted** node $x(j)$ that minimize the following formula:

$$\text{W(x(j))} + \alpha * \text{dist(x(i), x(j))} \text{ with } 0 \leq \text{j} < \text{i}$$

where:

- $W(x(j))$ is the distance in terms of hops (the path distance from node $x(j)$ to the root node);
- $dist(...)$ is the usual Euclidean distance;
- α is a weight parameter that can minimize or maximize the geometric distance influence;

After having chosen a node $x(j)$, we set $W(x(i)) = W(x(j)) + 1$. At the end we obtain a tree rooted in $x(0)$.

We have extended this model to improve robustness allowing every node to have exactly d outbound neighbors instead of only one. This means that at the time of joining the network, each node should have at least d candidates to be selected as neighbors. To achieve this property, as a first step we select at random exactly d root nodes and we connect them together in a ring fashion (a doubly linked list). In this way each ordinary node has at least d nodes (the d roots) to choose from in order to select its neighbors; in other words, each node has to select d nodes that minimize the function above.

To get further details about this model, we suggest the following readings:

1. "Heuristically Optimized Trade-offs: A New Paradigm for Power Laws in the Internet" (`http://cgi.di.uoa.gr/~elias/publications/paper-fkp02.pdffff.ps`)
2. "Degree distributions of the FKP network model" (`http://research.microsoft.com/~jchayes/Papers/FKPdgrees.pdf`)
3. "On Power-Law Relationships of the Internet Topology" (`http://www.cs.ucr.edu/~michalis/CAMERA.ps`)

The model should generate a topology that exhibits a power-law bound on the in-degree sequence of nodes; but, as stated in the second previously listed paper, this power-law prediction is not true.

B.3 What we need to code

In order to run this model in peersim we need to write java classes extending some peersim typical interfaces. Very in brief, we need:

- the protocol class: the protocol itself does nothing because we want something that automagically builds this topology model from a raw list of unconnected nodes; we are not interested in running any piece of code over time (cycles in peersim terminology). This class is a sort of structure to collect some needed values such as the node space coordinates, the hop distance, the in degree counter and so on. The reader can think to this class as "glue code".

- the initializer class: it extends the *Initializer* interface and deal with all the initialization process. As we'll see further, the initializer code itself is very compact because all the building process complexity is hidden in a custom made factory pattern class. Please note that we don't need explicitly an *Observer* object because we are not interested in observing any behavior over time; due to that, the code to track down informations about the actual generated tree (coordinates and indegree distribution) are embedded into the initializer.

B.4 Code writing

B.4.1 Protocol class

As we stated so far, the protocol code is minimal:

```
package inet;

import peersim.core.IdleProtocol;

public class InetNodeProtocol extends IdleProtocol {
    // coordinates in space:
    public double x;
    public double y;

    public int in_degree;
    public int hops;
    public boolean isroot;

    /** Creates a new instance of hotNodeProtocol */
    public InetNodeProtocol(String prefix, Object obj) {
   super(prefix);
        in_degree = 0;
        hops = 0;
        isroot = false;
    }

    public Object clone() throws CloneNotSupportedException {
        InetNodeProtocol af = (InetNodeProtocol) super.clone();
        ...
return af;
    }

    public void nextCycle(peersim.core.Node, int protocolID) {
    }
```

The *nextCycle()* method is empty, so it's presence is completely optional. The class is basically a structure encapsulated in an object.

B.4.2 Initializer class

This initializer can be considered as a public interface to the model. From the outside (means from the peersim configuration file) only few main parameters are needed, such as the *Linkable* enabled protocol, the outbound degree and the α parameter. The constructor method takes care of collecting these parameters or to set up the corresponding default values The actual structure generation is performed elsewhere.

```
public class InetInitializer implements peersim.init.Initializer {
     /**
      * String name of the parameter that defines the protocol to initialize.
     * Parameter read will has the full name
     * <tt>prefix+"."+PAR_PROT</tt>
     */
     public static final String PAR_PROT = "protocol";

     /**
      * String name of the parameter about the out degree value.
     */
     public static final String PAR_OUTDEGREE = "d";

     /**
      * String name of the parameter used as a weight.
     */
     public static final String PAR_ALFA = "alfa";

     /**
      * String name of the parameter used as a maximum x or y coordinate. All the
      * nodes are on a square region.
     */
     public static final String PAR_MAX_COORD = "max_coord";

     ....

     /** Creates a new instance of InetInitializer */
     public InetInitializer(String prefix) {
         // super(prefix);
         pid = Configuration.getInt(prefix+"."+PAR_PROT);
         d = Configuration.getInt(prefix+"."+PAR_OUTDEGREE);
         alfa = Configuration.getDouble(prefix+"."+PAR_ALFA);
         graph_filename = "cmplxnet_d"+ d + "_alfa"+alfa+".dat";
         dg_filename = "degree_d"+ d + "_alfa"+alfa+".dat";

         maxcoord = Configuration.getDouble(prefix + "." + PAR_MAX_COORD, 1.0);
         if ( !graph_filename.equals("") ) {
             try { graph_fileout = new PrintWriter(new FileWriter(graph_filename));
                 System.out.println(prefix +" filename: "+graph_filename +" selected");}
             catch (Exception e) {;}
         }
         if ( !dg_filename.equals("") ) {
             try { dg_fileout = new PrintWriter(new FileWriter(dg_filename));
                 System.out.println(prefix +" filename: "+dg_filename +" selected");}
             catch (Exception e) {;}
         }

     }
}
```

The *initialize()* method is defined by the *Initializer* interface; it invokes a specialized factory object that works on top of a *OverlayGraph* type object (actual implementation of the *Graph* interface). This object provides the high level abstraction of a graph on the simulator overlay

network; in this way, it allows the application to use many well known graph algorithms and operations (many operations can be found in *peersim.graphGraphAlgorithms* class). The operations performed on it are reflected on the actual topology (ex: adding an edge).

```
    public void initialize() {
        OverlayGraph ogr = new OverlayGraph(pid);
        InetFactory.InetTree(ogr, CommonRandom.r, pid, maxcoord, d, alfa );
        graphToFile(ogr);
        dgDistribToFile(ogr);
        }
    }
```

The other two method invocation in the *initialize()* method are used to write data on disk. In the first one, for each node n the x and y coordinates are collected and then for each neighbor i of node n the coordinates are written in the following order:

```
n.neighbor(i).x n.neighbor(i).y \newline
n.x n.y \newline
\newline}
```

The particular line triplet formatting order suits the gluplot needs. Please note that the for loop starts from index d, not from 0; this means that the root node(s) is not directly considered because it has not any outbound connections, but only inbound connections. Nevertheless the root node(s) are plotted inspecting the nodes that are linked directly to it.

```
    private void graphToFile(peersim.graph.Graph g) {
        if (graph_fileout != null) {
            try {
                for (int i = d ; i < g.size() ; i++ ) {
                    Node current = (Node)g.getNode(i);
                    double x_to = ((InetNodeProtocol)current.getProtocol(pid)).x;
                    double y_to = ((InetNodeProtocol)current.getProtocol(pid)).y;
                    Iterator it = (Iterator)g.getNeighbours(i).iterator();
                    while (it.hasNext()) {
                        int index = ((Integer)it.next()).intValue();
                        Node n = (Node)g.getNode(index);
                        double x_from = ((InetNodeProtocol)n.getProtocol(pid)).x;
                        double y_from = ((InetNodeProtocol)n.getProtocol(pid)).y;

                        graph_fileout.println(x_from+" "+y_from);
                        graph_fileout.println(x_to+" "+y_to);
                        graph_fileout.println("");
                    }
                }
                graph_fileout.close();
            }
            catch (Exception e) {;}
        }
    }
```

The second data collecting method builds an array of indegree frequencies and an array of indegree probability and dumps the second collection to file.

```
private void dgDistribToFile(peersim.graph.Graph g) {
        if (dg_fileout != null) {
            int size = g.size();
```

```
            try {
                int[] dgfrq = new int[size];
                double[] dgprob = new double[size];
                for (int i = 0 ; i < size ; i++) { // do not plot leaves
                    Node n = (Node)g.getNode(i);
                    InetNodeProtocol protocol = (InetNodeProtocol)n.getProtocol(pid);
                    int degree = protocol.in_degree;
                    dgfrq[degree]++;
                }
                double sum = 0;
                for (int i = size-1 ; i > 0 ; i--) {
                    dgprob[i] = (dgfrq[i] + sum)/size;
                    sum += dgfrq[i];
                }
                // do not count index 0: 'cos the leafs degree is clearly 0!
                for (int i = 0 ; i < dgprob.length ; i++ ) {
                    double k = (double)i/size;
                    dg_fileout.println(k+" "+dgprob[i]);
                }
                dg_fileout.close();
            }
            catch (Exception e) {e.printStackTrace();
                System.out.println(e);}
        }
    }
}
```

B.5 Factory class

This class is the core one. The actual topology initialization is performed here. Because of the factory pattern, al the methods in this class are static. There is no need of getting parameters from the configuration file, because they have been already collected by the initializer class; thus the factory is completely hidden.
This implementation is an extention of the peersim standard topology factory (*peersim.graph.Graph Factory*).

The only public method is the one that actually builds the topology: *InetTree()*; as parameters it gets all the parameters that the initializer class has collected. The steps performed are the following:

1. set the correct values for the d (at least one!) roots, including coordinates; if there is only one root, its coordinates are centered on the square (edge size 1.0 by default), otherwise the coordinates are random (as any ordinary node).

2. initialize the coordinates and the indegree counter for each ordinary node.

3. if there are more than one root node, than these root nodes are joined together in a ring (connections are non oriented). As an exercise, the reader can change this choice implementing something else (ex: putting the root nodes in a fully connected topology or a random graph or whatever).

4. for each node n other than the root, take exactly d nodes that minimizes the formula and connect node n to those d nodes.

The other present methods are all private and can be considered as utility methods. In fact their function is quite straightforward and can be guessed by the method signature; nevertheless a few comments are presented in the following table:

getParents()	get the the current node best d candidates to connect to
hops()	return the graph distance in terms of hops from the root of the node given as a parameter
minHops()	return the minimum hop valued node between the specified nodes
distance()	get the standard Euclidean distance between two nodes

```
package inet;

 import hot.HotNodeProtocol;
 import peersim.graph.*;
 import peersim.core.Node;
 import peersim.core.Linkable;
 import peersim.core.Network;
 import peersim.config.Configuration;
 import java.util.Random;
 import java.util.ArrayList;
 import java.util.Arrays;

 public class InetFactory extends peersim.graph.GraphFactory {

     private static final String DEBUG_STRING = "inet.InetFactory: ";

     public InetFactory() {
         super();
     }

     public static Graph InetTree(Graph g, Random rnd, int pid, double maxcoord,
     int outdegree, double alfa) {
         int size = g.size(); // size of the network
         System.out.println(DEBUG_STRING+"size: "+size+" outdegree: "+outdegree);

         // build outdegree roots
         System.out.println(DEBUG_STRING+"Generating "+outdegree+" root(s),
 means out degree "+outdegree+"...");
         for(int i = 0 ; i < outdegree ; ++i) {
             Node n = (Node)g.getNode(i);
             HotNodeProtocol prot = (HotNodeProtocol)n.getProtocol(pid);
             prot.isroot = true;
             prot.hops = 0;
             prot.in_degree = 0;
             if (outdegree == 1 ) {
                 prot.x = maxcoord/2;
                 prot.y = maxcoord/2;
             }
             else { // more than one root
                 if (rnd.nextBoolean() ) {
                     prot.x = maxcoord/2 + (rnd.nextDouble() * 0.1);
                 }
                 else {
                     prot.x = maxcoord/2 - (rnd.nextDouble() * 0.1);
                 }
                 if (rnd.nextBoolean() ) {
                     prot.y = maxcoord/2 + (rnd.nextDouble() * 0.1);
                 }
                 else {
                     prot.y = maxcoord/2 - (rnd.nextDouble() * 0.1);
```

```
        }
        System.out.println("root coord: "+prot.x+" "+prot.y);
    }
}

// Set coordinates x,y and set indegree 0
System.out.println(DEBUG_STRING+"Generating random cordinates for nodes...");
for (int i = outdegree ; i < size ; i++) {
    Node n = (Node)g.getNode(i);
    HotNodeProtocol prot = (HotNodeProtocol)n.getProtocol(pid);
    if (maxcoord == 1.0) {
        prot.x = rnd.nextDouble();
        prot.y = rnd.nextDouble();
    }
    else {
        prot.x = rnd.nextInt((int)maxcoord);
        prot.y = rnd.nextInt((int)maxcoord);
    }
    prot.in_degree = 0;
}

// Connect the roots in a ring if needed (thus, if there are more than 1
// root nodes.
if (outdegree > 1) {
    System.out.println(DEBUG_STRING+"Putting roots in a ring...");
    for (int i = 0 ; i < outdegree ; i++) {
        Node n = (Node)g.getNode(i);
        ((HotNodeProtocol)n.getProtocol(pid)).in_degree++;
        n = (Node)g.getNode(i+1);
        ((HotNodeProtocol)n.getProtocol(pid)).in_degree++;

        g.setEdge(i, i+1);
        g.setEdge(i+1, i);
    }
    Node n = (Node)g.getNode(0);
    ((HotNodeProtocol)n.getProtocol(pid)).in_degree++;
    n = (Node)g.getNode(outdegree);
    ((HotNodeProtocol)n.getProtocol(pid)).in_degree++;
    g.setEdge(0, outdegree);
    g.setEdge(outdegree, 0);
}

// for all the nodes other than root(s), connect them!
for (int i = outdegree ; i < size ; ++i ) {
    Node n = (Node)g.getNode(i);
    InetNodeProtocol prot = (InetNodeProtocol)n.getProtocol(pid);

    prot.isroot = false;

    // look for a siutable parent node between those allready part of the
    // overlay topology: alias FIND THE MINIMUM!
    Node candidate = null;
    int candidate_index = 0;
    double min = Double.POSITIVE_INFINITY;
    if (outdegree > 1) {
            int candidates[] = getParents(g, pid, i, outdegree, alfa);
            for (int s = 0 ; s < candidates.length ; s++) {
                g.setEdge(i, candidates[s]);
                System.out.print(i+", ");
            }
            prot.hops = minHop(g, candidates, pid) + 1;
    }
    else { // degree 1:
        for (int j = 0 ; j < i ; j++) {
            Node parent = (Node)g.getNode(j);
```

```
                InetNodeProtocol prot_parent = (InetNodeProtocol)parent.getProtocol(pid);

                double value = hops(parent, pid) +
                    (alfa * distance(n, parent, pid));
                if (value < min) {
                    candidate = parent; // best parent node to connect to
                    min = value;
                    candidate_index = j;
                }
            }
            prot.hops = ((InetNodeProtocol)candidate.getProtocol(pid)).hops + 1;
            g.setEdge(i, candidate_index);
            ((HotNodeProtocol)candidate.getProtocol(pid)).in_degree++;
        }

    }
    System.out.println(DEBUG_STRING+"Graph generation finished!");
    return g;
}

private static int[] getParents(Graph g, int pid, int cur_node_index,
int how_many, double alfa) {
    int result[] = new int[how_many];
    ArrayList net_copy = new ArrayList(cur_node_index);
    // fill up the sub net copy:
    for (int j = 0 ; j < cur_node_index ; j++) {
        net_copy.add(j, (Node)g.getNode(j));
    }

    // it needs exactly how_many minimums!
    for (int k = 0 ; k < how_many ; k++) {
        int candidate_index = 0;
        double min = Double.POSITIVE_INFINITY;
        // for all the elements in the copy...
        for (int j = 0 ; j < net_copy.size() ; j++) {
            Node parent = (Node)net_copy.get(j);
            HotNodeProtocol prot_parent = (HotNodeProtocol)parent.getProtocol(pid);
            double value = hops(parent, pid) + (alfa *
                distance((Node)g.getNode(cur_node_index), parent, pid));

            if (value < min) {
                min = value;
                candidate_index = j;
            }
        }
        result[k] = candidate_index;        // collect the parent node
        net_copy.remove(candidate_index); // delete the min from the net copy
    }
    return result;
}


private static int hops(Node node, int pid) {
    return ((HotNodeProtocol)node.getProtocol(pid)).hops;
}

private static int minHop(Graph g, int[] indexes, int pid) {
    int min = Integer.MAX_VALUE;
    for (int s = 0 ; s < indexes.length ; s++) {
        Node parent = (Node)g.getNode(indexes[s]);
        int value = ((HotNodeProtocol)parent.getProtocol(pid)).hops;
        if (value < min) {
            min = value;
        }
    }
```

```
            return min;
        }

        private static double distance(Node new_node, Node old_node, int pid) {
            // Eucleidian distance code...
        }
    }
```

B.6 Experiments

In order to make the model run, a proper peersim configuration file is needed. The one presented in the following lines may suits the reader needs:

```
# Complex Network file:
random.seed 1234567890
simulation.cycles 1

overlay.size 10000
overlay.maxSize 10000

protocol.0 hot.InetNodeProtocol
#protocol.0.maxcoord 1000

init.0 hot.InetInitializer
init.0.alfa 20
init.0.protocol 0
init.0.d 1
```

It produces a 10000 node overlay network with the parameters listed in the `init.0` section. The figures in the table shows the produced topology and highlights the parameter α importance. In fact it affects the clustering behavior of the system and it is tightly correlated to the size of the network. If α il lower than $\sqrt{netsize}$, the topology becomes more and more clusterized (as show in the first two figures); with extremely low α, the topology becomes a star. On the other end, if α is grater than $\sqrt{netsize}$, the topology tends to be random and not clusterized at all (the second row of images). For deeper details, please consult the previously listed papers.

All the images has been produced using only one root node and only one outbound connection per node. Using two or more outbound connection per node leads to a massively crowded plot, so it's not a very nice picture to plot!

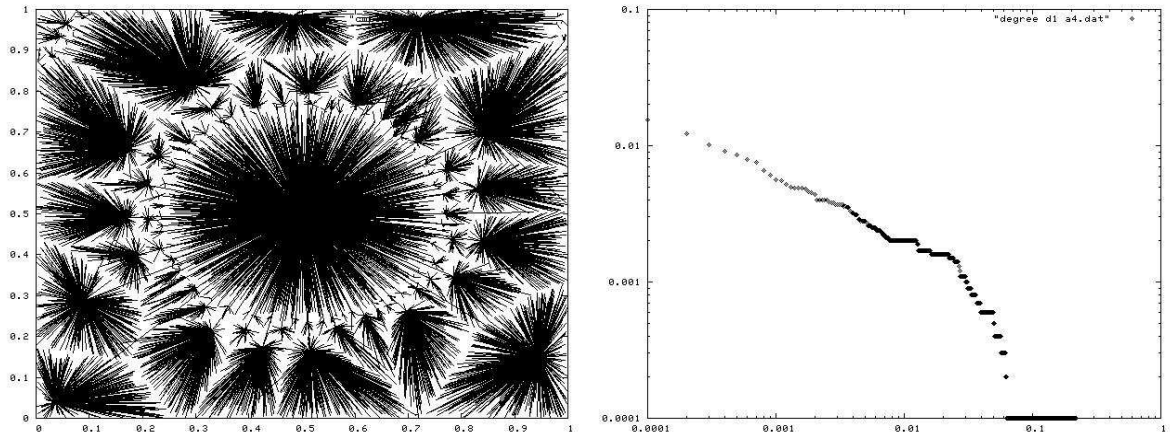

Figure 7: Topology and indegree distribution with α 4

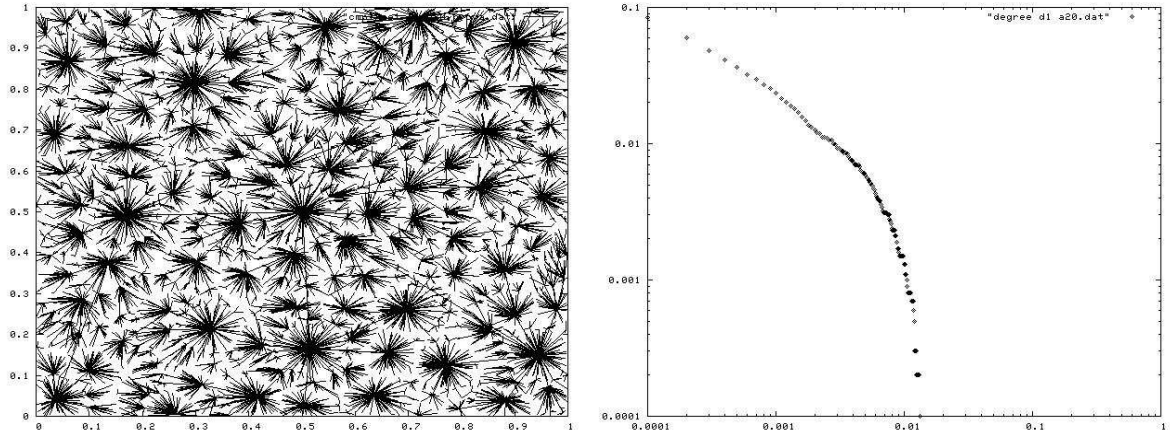

Figure 8: Topology and indegree distribution with α 20

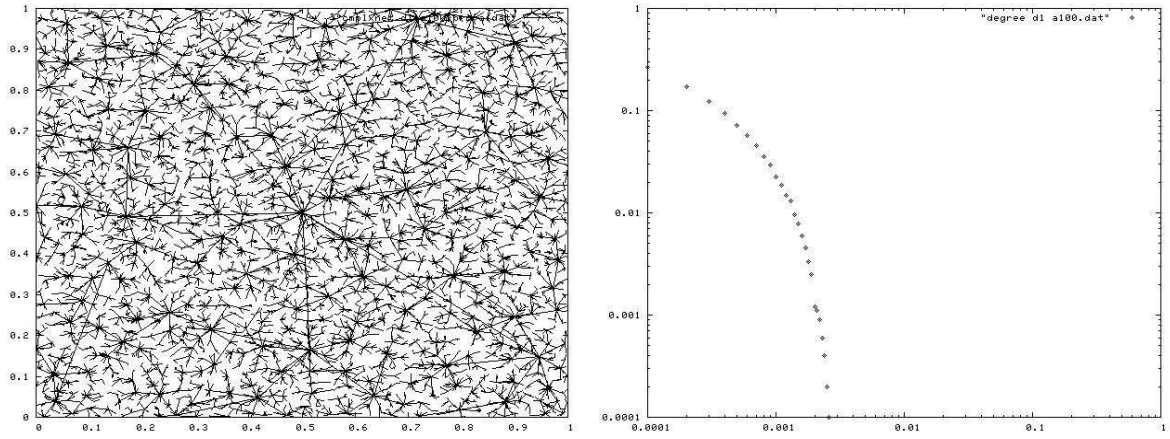

Figure 9: Topology and indegree distribution with α 100