

PeerSim HOWTO:

Build a new protocol for the PeerSim 1.0 simulator

Gian Paolo Jesi (jesi@cs.unibo.it)

December 24, 2005

This tutorial is for PeerSim 1.0, in particular, its so-called cycle based engine. If the reader is not a PeerSim newbie, she may be interested in what major changes have been introduced since the last version. The changes are summarized in Appendix C. For a more detailed list, please check the PeerSim CHANGELOG file, included in the distribution.

1 Introduction

This tutorial is a step by step guide to build from scratch a new PeerSim application. In this tutorial it is supposed that you have access to:

- knowledge of the Java language
- a Java compiler for Java version 1.5; we propose \geq JDK 1.5.x
- the PeerSim 1.0 package (<http://sourceforge.net/projects/PeerSim>), or the PeerSim source tree (you can download it from the sourceforge CVS server)
- the Java Expression Parser version \geq 2.3.0 (included in the release, but you can download it from <http://www.singularsys.com/jep/>)
- the Gnuplot plotting tool is highly recommended

The aim of this tutorial is to be as practical and lightweight as possible. The goal is to give the reader the basics of PeerSim and a step-by-step example about writing simple protocols in PeerSim. This tutorial IS NOT exhaustive, it is not meant to be a comprehensive guide, but it is enough to get you started.

2 Introduction to PeerSim

2.1 Why PeerSim

Peer-to-peer (P2P) systems can be extremely large scale (millions of nodes). Nodes in the network join and leave continuously. Experimenting with a protocol in such an environment it no easy task at all.

PeerSim has been developed to cope with these properties and thus to reach extreme scalability and to support dynamism. In addition, the simulator structure is based on components

and makes it easy to quickly prototype a protocol, combining different pluggable building blocks, that are in fact Java objects.

PeerSim 1.0 supports two simulation models: the *cycle-based* model and a more traditional *event-based* model. This document is about the former. This model is a simplified one, which makes it possible to achieve extreme scalability and performance, at the cost of some loss of realism. Several simple protocols can tolerate this loss without problems, according to our own experience; yet, care should be taken when this model is selected to experiment with a protocol.

The simplifying assumptions of the cycle based model are the lack of transport layer simulation and the lack of concurrency. In other words, nodes communicate with each other directly, and the nodes are given the control periodically, in some sequential order, when they can perform arbitrary actions, such as call methods of other objects and perform some computations.

We note here that it is relatively easy to migrate any cycle-based simulations to the event driven engine. However, we don't discuss how to do it in this document.

2.2 PeerSim simulation life-cycle

PeerSim was designed to encourage modular programming based on objects (building blocks). Every block is easily replaceable by another component implementing the same interface (i.e., the same functionality). The general idea of the simulation model is:

1. choose a network size (number of nodes)
2. choose one or more protocols to experiment with and initialize them
3. choose one or more *Control* objects to monitor the properties you are interested in and to modify some parameters during the simulation (e.g., the size of the network, the internal state of the protocols, etc)
4. run your simulation invoking the *Simulator* class with a configuration file, that contains the above information

All the object created during the simulation are instances of classes that implement one or more interfaces. The main interfaces to become familiar with are listed in Table 1.

The life-cycle of a cycle-based simulation is as follows. The first step is to read the configuration file, given as a command-line parameter (see Section 2.3). The configuration contains all the simulation parameters concerning all the objects involved in the experiment.

Then the simulator sets up the network initializing the nodes in the network, and the protocols in them. Each node has the same kinds of protocols; that is, instances of a protocols form an array in the network, with one instance in each node. The instances of the nodes and the protocols are created by cloning. That is, only one instance is constructed using the constructor of the object, which serve as prototypes, and all the nodes in the network are cloned from this prototype. For this reason, it is very important to play attention to the implementation of the cloning method of the protocols.

At this point, initialization needs to be performed, that sets up the initial states of each protocol. The initialization phase is carried out by *Control* objects that are scheduled to run only at the beginning of each experiment. In the configuration file, the initialization

<i>Node</i>	The P2P network is composed of nodes. A node is a container of protocols. The node interface provides access to the protocols it holds, and to a fixed ID of the node.
<i>CDProtocol</i>	It is a specific protocol, that is designed to run in the cycle-driven model. Such a protocol simply defines an operation to be performed at each cycle.
<i>Linkable</i>	Typically implemented by protocols, this interface provides a service to other protocols to access a set of neighbor nodes. The instances of the same linkable protocol class over the nodes define an overlay network.
<i>Control</i>	Classes implementing this interface can be scheduled for execution at certain points during the simulation. These classes typically observe or modify the simulation.

Table 1: Suggested PeerSim subset of classes or interfaces to know about.

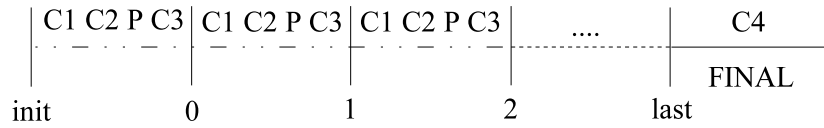


Figure 1: Scheduling controls and protocols. The “C” letters indicate a control component, while letter “P” indicates a protocol. The numbers in lower part of the picture indicate the PeerSim cycles. After the last cycle, it is possible to run a final control to retrieve a final snapshot.

components are easily recognizable by the `init` prefix. Please note that in the following pages we will talk about *Initializer* objects just to remark their function and to distinguish them from ordinary *Control* objects, but the initializer objects are simply controls, configured to run in the initialization phase.

After initialization, the cycle driven engine calls all components (protocols and controls) once in each cycle, until a given number of cycles, or until a component decides to end the simulation. Each object in PeerSim (controls and protocols) is assigned a *Scheduler* object which defines when they are executed exactly. By default, all objects are executed in each cycle. However, it is possible to configure a protocol or control to run only in certain cycles, and it is also possible to control the order of the running of the components within each cycle. The latter scenario is illustrated in Figure 1.

When a *Control* has to collect data, they are formatted and sent to standard output and can be easily redirected to a file to be collected for further work.

2.3 The configuration file

The configuration file is a plain ASCII text file, basically composed of key-value pairs representing Java *java.util.Properties*; the lines starting with “#” character are ignored (comments).

The config file is specified on the command line as follows:

```
java peersim.Simulator config-file.txt
```

2.4 Configuration example 1

We are going to create a fixed P2P random topology composed of 50000 nodes. The chosen protocol is *Aggregation* (what is aggregation? see appendix A) using the average function. The values to be aggregated (averaged) at each node are initialized using a linear distribution on the interval (0, 100). Finally a *Control* monitors the averaging values. Looks easy!

```
01 # PEERSIM EXAMPLE 1
02
03 random.seed 1234567890
04 simulation.cycles 30
05
06 control.shf Shuffle
07
08 network.size 50000
09
10 protocol.lnk IdleProtocol
11
12 protocol.avg example.aggregation.AverageFunction
13 protocol.avg.linkable lnk
14
15 init.rnd WireKOut
16 init.rnd.protocol lnk
17 init.rnd.k 20
18
19 init.peak example.aggregation.PeakDistributionInitializer
20 init.peak.value 10000
21 init.peak.protocol avg
22
23 init.lin LinearDistribution
24 init.lin.protocol avg
25 init.lin.max 100
26 init.lin.min 1
27
28 # you can change this to select the peak initializer instead
29 include.init rnd lin
30
31 control.avgo example.aggregation.AverageObserver
32 control.avgo.protocol avg
```

The first things to note are the key names: some of them refer to global properties, while some others refer to single component instances. For example, `simulation.cycles` is global, but `protocol.lnk.xxx` defines parameter `xxx` of protocol `lnk`.

Observe that each component has a name, such as `lnk`. In the case of protocols, this name is mapped to a numeric index called protocol ID, by the PeerSim engine. This index does not appear in the configuration file, but it is necessary to access protocols during a simulation. We give more details later.

A component such as a protocol or a control is declared by the following syntax:

```
<protocol|init|control>.string_id [full_path_]classname
```

Note that the full class path is optional, in fact PeerSim can search in its classpath in order to find a class. If multiple classes share the same name (in distinct packages), the full path

is needed. The prefix `init` defines an initializer object, that has to implement the Control interface.

The component parameters (if any) follows this scheme:

```
<protocol|init|control>.string_id.parameter_name
```

For example, at **line 10**, the first protocol chosen comes to life; the **key part** contains its type, in this case `protocol` followed by the name, in this case `lnk`, and the **value part** contains the classname for the component, in this case `IdleProtocol`. This class is in the `peersim` package, and you don't have to know what is its fully specified name.

Parameters can be declared for each component. For example, see **line 13**, where the **key part** contains the parameter name and the **value part** is simply the value desired.

From **line 3 to line 8** some global simulation properties are imposed; these are the total number of simulation cycles and the overlay network size. The *Shuffle* control (**line 6**) shuffles the order in which the nodes are visited in each cycle.

From **line 10 to line 13**, two protocols are put in the arena. The first one, *IdleProtocol* does nothing. It simply serves as a static container of links to neighboring nodes. The class *IdleProtocol* does not implement *CDProtocol*, but it does implement the *Linkable* interface, through which it provides the links to neighbors.

The second protocol (`protocol.avg aggregation.AverageFunction`) is the averaging version of aggregation. Its parameter (`linkable`) is important: the aggregation protocol needs to talk to neighbors but doesn't have its own list of neighbors. In a modular fashion, it can be put over any overlay network. The protocol (array) that defines the overlay network has to be specified here. The value of parameter `linkable` is the name of a protocol implementing the *Linkable* interface (*IdleProtocol* in the example).

From **line 15 to line 26**, it is time to initialize all the components previously declared. We declare three initialization components, but only two of them are actually used (see **line 29**). The first initializer, `peersim.init.WireKOut`, performs the wiring of the static overlay network. In particular, the nodes are linked randomly to each-other to form a random graph having the specified degree (`k`) parameter.

The second and third initializers are alternatives to initialize the aggregation protocol, in particular, the initial values to be averaged. The initializers set the initial values to follow a peak or linear distribution, respectively. Peak means that only one node will have a value different from zero. Linear means that the nodes will have linearly increasing values. Both initializers need a protocol name that identifies the protocol to initialize (`protocol` parameter). Additional parameters are the range (`max`, `min` parameters) for the *PeakDistributionInitializer* and `value` parameter for *LinearDistribution*.

The choice to use the peak or linear distribution is given by the `include.init` property (**line 29**) that selects which initializers are allowed to run. This property also defines the order in which the components are run. Note that the default order (if there is no `include` property) is according to alphabetical order of names. A similar `include` property works also with protocols and controls.

Finally at **line 31, 32** the last component is declared: `aggregation.AverageObserver`. Its only parameter used is `protocol` which refers to the *aggregation.AverageFunction* protocol type, so the parameter value is `avg`.

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

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

The class-path is mandatory only if the current 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 3**:

```
random.seed 1234567890
```

on top of the configuration file. This parameter is very useful to replicate exactly the experiment results based on (pseudo) random behavior. The standard output is:

```
control.avgo: 0 1.0 100.0 50000 50.49999999999999 816.7990066335468 1 1
control.avgo: 1 1.2970059401188023 99.38519770395408 50000 50.500000000000005 249.40673287686545 1 1
control.avgo: 2 9.573571471429428 84.38874902498048 50000 50.500000000000085 77.89385877895182 1 1
control.avgo: 3 23.860361582231647 71.93627224106982 50000 50.499999999999967 24.131366707228402 1 1
control.avgo: 4 34.920915967147465 68.92828482118958 50000 50.499999999999994 7.702082905414273 1 1
control.avgo: 5 42.37228198409946 59.94511004870823 50000 50.499999999999987 2.431356211088775 1 1
control.avgo: 6 45.19621912151794 54.855516163070746 50000 50.499999999999844 0.7741451706754877 1 1
control.avgo: 7 47.68716274528092 53.11433934745646 50000 50.499999999999949 0.24515365729069857 1 1
control.avgo: 8 48.97706271318158 52.38916238021276 50000 50.500000000000026 0.07746523384731269 1 1
control.avgo: 9 49.59674440194668 51.46963472637451 50000 50.499999999999937 0.024689348817011823 1 1
control.avgo: 10 49.946490417215266 51.13343750384934 50000 50.500000000000048 0.007807022577928414 2 1
control.avgo: 11 50.18143472395333 50.858337267869565 50000 50.499999999999982 0.002493501256296898 2 1
control.avgo: 12 50.30454978101492 50.67203454827276 50000 50.5000000000000206 7.90551008686205E-4 1 1
control.avgo: 13 50.3981394834783 50.60093898689035 50000 50.499999999999967 2.518940347803474E-4 1 1
control.avgo: 14 50.449347314832124 50.54962989951735 50000 50.500000000000003 8.071623184942779E-5 1 1
control.avgo: 15 50.47368195506415 50.52608817343459 50000 50.499999999999999 2.566284350168338E-5 1 1
control.avgo: 16 50.48510475374435 50.518871021756894 50000 50.500000000000012 8.191527862075119E-6 1 1
control.avgo: 17 50.49082426764112 50.51000681641142 50000 50.499999999999945 2.570199757692886E-6 1 1
control.avgo: 18 50.494810505765045 50.50556221303088 50000 50.500000000000003 8.197012224814065E-7 1 1
control.avgo: 19 50.496876367842034 50.50296444951085 50000 50.4999999999999524 2.640584231868471E-7 1 1
control.avgo: 20 50.498457906558905 50.50182062146254 50000 50.5000000000000334 8.565428611988968E-8 1 1
control.avgo: 21 50.49905541635283 50.50096466374638 50000 50.499999999999974 2.721171621666857E-8 1 1
control.avgo: 22 50.49946061473347 50.500553628252945 50000 50.499999999999975 8.590349265230611E-9 1 1
control.avgo: 23 50.49972602272376 50.500315571370415 50000 50.500000000000004 2.6248542064007986E-9 2 1
control.avgo: 24 50.4998450606816 50.50018053311878 50000 50.500000000000005 8.845012874999227E-10 1 1
control.avgo: 25 50.499894793874255 50.500096923965216 50000 50.500000000000079 1.864501428663076E-10 1 2
control.avgo: 26 50.4999267984512 50.500056126785694 50000 50.500000000000003 8.594896829690765E-11 1 1
control.avgo: 27 50.49996613170552 50.50003198608762 50000 50.500000000000017 1.9554527178661528E-11 1 1
control.avgo: 28 50.49997903068333 50.500019172164286 50000 50.499999999999766 3.274246411310768E-11 1 1
control.avgo: 29 50.49998958653935 50.5000099409645 50000 50.500000000000045 0.0 1 1
```

The observer component produces many numbers, but looking at the 3th and 4th data columns (the maximum and the minimum of the values over the network) it is easy to see how the variance decreases very quickly. At cycle 12, quite all the nodes have a very good approximation of the real average (50). Try to experiment with different numbers or change the init distribution (e.g.: using `aggregation.PeakDistributionInitializer`). You can also replace the overlay network, for example, you can configure *Newscast* instead of *IdleProtocol* as it is done in the next example.