A Basic Event Driven Example for PeerSim 1.0

Márk Jelasity

November 18, 2006

1 Introduction

This document walks you through a basic example application of PeerSim, that uses the *event based* simulation model. We assume that you are familiar with the examples described in the short tutorials on the cycle based model: the basic working of PeerSim, its configuration file, and the gossip-based averaging example, where nodes collectively calculate the average of some parameter via periodically exchanging messages and performing pairwise averaging steps. The example discussed here will be the same: gossip-based averaging. Only this time using the event based model, where message sending is modeled in more detail. This will allow us to observe problems with the protocol that remained hidden in the cycle based simulations.

In the event based model, everything works exactly the same way as in the cycle based model, except time management and the way control is passed to the protocols. *Protocols* that are not executable (that are used only to store data, such as some linkable protocols that only store neighbors, or vectors that store numeric values) can be applied and initialized in exactly the same way. *Controls* in any package outside package *peersim.cdsim* can be used as well. By default controls are called in each cycle in the cycle based model. In the event based model they have to be scheduled explicitly, since there are no cycles.

Evidently, we can write controls that are specific to the event based model, that is, that are able to send events (messages) to protocols. In many cases, this will be necessary because often the system will be driven completely or partially by external events such as queries by users, that can best be modeled by controls that generate these events and thereby drive the execution of the simulation.

There are components that are not available. This includes any component that relies on the static class *peersim.cdsim.CDState* which offers an interface to read cycle-specific global state. Our experience is that many components written for the cycle based model that rely on this state can be easily modified to remove this dependency.

However, maybe a bit surprisingly, protocols that implement the cycle based interface *peersim.cdsim.CDProtocol* can be utilized in the event based model as well; we will explain later how. But it should be noted immediately that this must be done with care, because in most of the cases it does not make any sense. However, this feature does have its very useful applications: it makes it possible to easily invoke protocols periodically, a feature that is characteristic to practically all P2P protocols in connection with housekeeping, failure detection and sending heartbeat messages.

2 Event Based Averaging: the Protocol

Let us start with the Java class that implement our averaging protocol in the event based model, that begins like this:

```
package example.edaggregation;

import peersim.vector.SingleValueHolder;
import peersim.config.*;
import peersim.core.*;
import peersim.transport.Transport;
import peersim.cdsim.CDProtocol;
import peersim.edsim.EDProtocol;

/**
* Event driven version of epidemic averaging.
*/
public class AverageED extends SingleValueHolder
implements CDProtocol, EDProtocol {
```

The first thing we notice is that we implement interface *EDProtocol* and also *CDProtocol*. The former will allow the class to process incoming messages. The latter is a bit confusing, as it belongs to the cycle based model. First of all, protocols need not implement this interface in the event based model. However, protocols that would like to get control periodically can achieve this effect using this trick of implementing *CDProtocol*, and setting a *CDScheduler* in the configuration, as we will describe below. This has several advantages over fiddling with timers and so on. The code is much cleaner, the periodic execution that is managed by a separate component is modular itself and can be configured separately, and finally, we can port our old cycle based protocols much easier to the event based model.

```
/**
 * @param prefix string prefix for config properties
 */
public AverageED(String prefix) { super(prefix); }
```

Our simple protocol does not read any configuration parameters. Now let us turn to the implementation of the cycle based interface. This method defines the activity to be performed periodically.

```
/**
 * This is the standard method the define periodic activity.
 * The frequency of execution of this method is defined by a
 * {@link peersim.edsim.CDScheduler} component in the configuration.
 */
public void nextCycle( Node node, int pid )
{
        Linkable linkable =
                (Linkable) node.getProtocol( FastConfig.getLinkable(pid) );
        if (linkable.degree() > 0)
        {
                Node peern = linkable.getNeighbor(
                                CommonState.r.nextInt(linkable.degree()));

                // XXX quick and dirty handling of failures
                // (message would be lost anyway, we save time)
```

```
                if(!peern.isUp()) return;

                AverageED peer = (AverageED) peern.getProtocol(pid);

                ((Transport)node.getProtocol(FastConfig.getTransport(pid))).
                        send(
                                node,
                                peern,
                                new AverageMessage(value,node),
                                pid);
        }
}
```

What we need to observe here is the stuff specific to the event based model. This boils down to handling the transport layer. First of all, class *FastConfig* gives us a way to access the transport layer that was configured for this protocol. Using this transport layer, we can send messages to protocols on other nodes. A message can be an arbitrary object. Since the simulator is not distributed, there is no trouble with serialization, etc; the object will be stored by reference.

The target protocol is defined by the target node `peern`, and the protocol identifier of the target protocol `pid`. In our case, we send a message to the same protocol on a different node. Evidently, the target protocol has to implement the *EDProtocol* interface.

```
/**
* This is the standard method to define to process incoming messages.
*/
public void processEvent( Node node, int pid, Object event ) {

        AverageMessage aem = (AverageMessage)event;

        if( aem.sender!=null )
                ((Transport)node.getProtocol(FastConfig.getTransport(pid))).
                        send(
                                node,
                                aem.sender,
                                new AverageMessage(value,null),
                                pid);

        value = (value + aem.value) / 2;
}

}
```

The method above is specified in *EDSimulator* and is supposed to handle incoming messages. In our example, there is only one type of message. All we need to check whether the sender is null, because if it is, it means that we do not need to answer the message (it is already an answer). If we need to answer, we do this the same way as we have seen already, through the transport layer.

```
/**
* The type of a message. It contains a value of type double and the
* sender node of type {@link peersim.core.Node}.
*/
class AverageMessage {
```

```
        final double value;
        /** If not null,
        this has to be answered, otherwise this is the answer. */
        final Node sender;
        public AverageMessage( double value, Node sender )
        {
                this.value = value;
                this.sender = sender;
        }
}
```

This private class is the type of the message that the protocol uses. It is private because no other component has anything to do with the message type.

3 Event Based Averaging: the Configuration

Let us examine a configuration file that can be used to run the event based simulation. It is very similar to the cycle based configuration, the difference is slight, but important.

```
# network size
SIZE 1000

# parameters of periodic execution
CYCLES 100
CYCLE SIZE*10000

# parameters of message transfer
# delay values here are relative to cycle length, in percentage,
# eg 50 means half the cycle length, 200 twice the cycle length, etc.
MINDELAY 0
MAXDELAY 0
# drop is a probability, 0<=DROP<=1
DROP 0
```

We have just defined a number of constants to make the configuration file cleaner and easier to change from the command line. For example, `CYCLE` defines the length of a cycle.

```
random.seed 1234567890
network.size SIZE
simulation.endtime CYCLE*CYCLES
simulation.logtime CYCLE
```

Parameter `simulation.endtime` is the key thing here: it tells the simulator when to stop. The internal representation of time is long (64 bit integer). It is zero at startup time and it is advanced by message delays. Simulation stops when the event queue is empty (nothing left to do) or if all the events in the queue are scheduled for a time later than the specified end time.

The simulator prints indications on the standard error about the progress of time. Parameter `simulation.logtime` specifies the frequency of these messages.

```
################### protocols ===========================

protocol.link peersim.core.IdleProtocol

protocol.avg example.edaggregation.AverageED
protocol.avg.linkable link
protocol.avg.step CYCLE
protocol.avg.transport tr

protocol.urt UniformRandomTransport
protocol.urt.mindelay (CYCLE*MINDELAY)/100
protocol.urt.maxdelay (CYCLE*MAXDELAY)/100

protocol.tr UnreliableTransport
protocol.tr.transport urt
protocol.tr.drop DROP
```

Here we configure our protocol (`avg`) and specify the overlay network (`link`) and the transport layer (`tr`). We also have to specify the `step` scheduling parameter, familiar from the cycle based model. This is because we have implemented the cycle based interface, so we need to specify how long a cycle is, in order to be able to make use of it.

The overlay network is just a container of links that will remain constant throughout the simulation and that will be initialized as shown below.

The transport layer is also configured as a protocol. It models both random delays and message drops. First we define a transport layer that has random delay (`urt`) and then we wrap it in a generic wrapper that takes any transport layer, and drops messages with a given probability (`tr`). Transport layers are defined in package *peersim.transport*. As everything, this component is also modular and custom transport layers can be easily developed and used.

```
################### initialization ======================

init.rndlink WireKOut
init.rndlink.k 20
init.rndlink.protocol link

init.vals LinearDistribution
init.vals.protocol avg
init.vals.max SIZE
init.vals.min 1

init.sch CDScheduler
init.sch.protocol avg
init.sch.randstart
```

Here the only component that is specific to the event based model is `sch`. It is responsible for scheduling the periodic call of the cycle based interface (`nextCycle`). In this configuration, this component will first assign a random point in time between 0 and `CYCLE` to all nodes, which will be the first time `nextCycle` is called on protocol `avg`. Then the next calls will happen in intervals of exactly `CYCLE` time steps regularly. More advanced methods also exist, besides, the scheduling can be customized by class extension; we do not go into the details here.

```
################ control =============================
```

```
control.0 SingleValueObserver
control.0.protocol avg
control.0.step CYCLE
```

This we have seen already. Note that we need to specify the `step` parameter here as well, just like for protocol `avg`. This will specify how often this control will be called. Otherwise controls can be scheduled the same way as in the cycle based model, only there is no default `step`, because there are no cycles.

4 Running the Protocol

If we invoke the configuration file above, we should get the following on standard error:

```
Simulator: loading configuration
ConfigProperties: File config-edexample.txt loaded.
Simulator: starting experiment 0 invoking peersim.edsim.EDSimulator
Random seed: 1234567890
EDSimulator: resetting
Network: no node defined, using GeneralNode
EDSimulator: running initializers
- Running initializer init.rndlink: class peersim.dynamics.WireKOut
- Running initializer init.sch: class peersim.edsim.CDScheduler
- Running initializer init.vals: class peersim.vector.LinearDistribution
EDSimulator: loaded controls [control.0]
Current time: 0
Current time: 10000000
Current time: 20000000
Current time: 30000000
Current time: 40000000
Current time: 50000000
.
.
.
Current time: 980000000
Current time: 990000000
EDSimulator: queue is empty, quitting at time 999980413
```

and the following on standard output:

```
control.0: 1.0 1000.0 1000 500.5 83416.66666666667 1 1
control.0: 37.5 919.0 1000 500.5 25724.159091250687 1 1
control.0: 206.7109375 767.890625 1000 500.5 8096.807036889389 1 1
control.0: 352.373046875 695.453125 1000 500.5 2578.022573176135 1 1
control.0: 412.430419921875 625.474609375 1000 500.5 801.1082179446831 1 1
control.0: 436.43787479400635 570.459858417511 1000 500.5 243.53994072762902 1 1
control.0: 470.7608990445733 527.0359845032217 1000 500.49999999999994 74.13788674564383 1 2
control.0: 483.6040476858616 518.0301055684686 1000 500.49999999999903 23.428974301677556 1 1
control.0: 490.5196089811798 512.0301471857779 1000 500.4999999999993 7.285566419597019 1 1
control.0: 494.97216907397836 506.0375954180854 1000 500.4999999999999 2.1798299307442246 1 1
control.0: 497.18190345272336 503.5837144460532 1000 500.5000000000001 0.6073148838336206 1 1
control.0: 498.54320551492475 502.3533156558903 1000 500.5 0.1786794435445898 1 2
control.0: 499.4023441821402 501.4962048486104 1000 500.49999999999966 0.055257607540637785 1 1
control.0: 500.0032071191514 501.09832936709677 1000 500.4999999999995 0.017914865984002482 1 1
.
.
.
control.0: 500.5 500.5 1000 500.5 0.0 1000 1000
```

```
control.0: 500.5 500.5 1000 500.5 0.0 1000 1000
control.0: 500.5 500.5 1000 500.5 0.0 1000 1000
control.0: 500.5 500.5 1000 500.5 0.0 1000 1000
```

Recall that the meaning of the values are min, max, number of samples, average, variance, number of instances of min, and number of instances of max in the sample. This output indicates that the correct average (500.5) is found, with zero variance (all nodes hold the correct average).

This seems nice, but since we can now play with delay, we can add some delay and see what happens (in the default config file the delay was zero). So, how about appending `MINDELAY=10 MAXDELAY=10` to the command line, which means that all messages will be delayed by exactly 10% of the cycle length. We get

```
.
.
.
control.0: 499.126081326076 499.126081326076 1000 499.12608132608807 0.0 1000 1000
control.0: 499.126081326076 499.126081326076 1000 499.12608132608807 0.0 1000 1000
control.0: 499.126081326076 499.126081326076 1000 499.12608132608807 0.0 1000 1000
```

That is, our simple delay scheme already destroys the nice properties of the protocol: we have convergence, but the result is incorrect. One can verify that different random seeds give different results, and changing the delay interval and drop rate also have their effects on performance.

So, how about keeping the delay and drop rate zero. Is that a guarantee that we get correct behavior? Not really. Let us experiment with a shorter cycle length, for example, `CYCLE=SIZE`. This means that there will often be more events scheduled to happen at the same time point. In such cases, PeerSim executes those events in a random order. We obtain:

```
.
.
.
control.0: 500.4835099381911 500.4835099381911 1000 500.48350993818605 7.807634196601234E-9 1000 1000
control.0: 500.4835099381911 500.4835099381911 1000 500.48350993818605 7.807634196601234E-9 1000 1000
control.0: 500.4835099381911 500.4835099381911 1000 500.48350993818605 7.807634196601234E-9 1000 1000
```

What is the conclusion? We must say this time the incorrect result is more due to an artifact, that is, insufficient time resolution. If messages indeed have zero delay, then even the slightest difference in execution time results in non-overlapping pairwise exchanges. Clearly, in continuous time no events will happen at the same time. However, the slightest random delay in message delivery renders the result meaningful, because then the order-uncertainty is indeed real.

All in all, we see that the event based simulator model can reveal problems with protocols not visible in the cycle based model, however, it also introduces new artifacts.

5 Disclaimer

This simple example is only to get you started. You have not seen all. It is highly recommended to study the class documentation of the relevant packages (*peersim.edsim, peersim.transport*) for maximal control, and it does not hurt if you are also familiar with the generic components of PeerSim as well.