PeerSim: Informal introduction

Alberto Montresor Gianluca Ciccarelli

Networking group - University of Trento

April 3, 2009

Outline

1. PeerSim's Interfaces
   - CDProtocol and EDProtocol
   - Cycle-based simulation
   - Event-driven simulation
2. Configuration
   - Principle
   - Configuration files' structure
   - Examples
3. Implementation
   - Example: Traffic injection
   - The Vector class

Resources

- These slides (and material presented during classes): `http://disi.unitn.it/~ciccarelli` (section: Teaching)
- Also see the document `projects.pdf`

What is PeerSim?

The PeerSim simulator is a simulation engine

- to write network protocols
- for P2P networks
- in a scalable and portable way
- Website of the project: http://peersim.sourceforge.net/

What is a simulation engine?

A simulation engine is an application on top of which (by means of which) we can write simulations, collect results and analyze them. The engine takes care of the temporal structure of the experiment, while the programmer takes care of the logic of the interactions among the elements of the scenario.

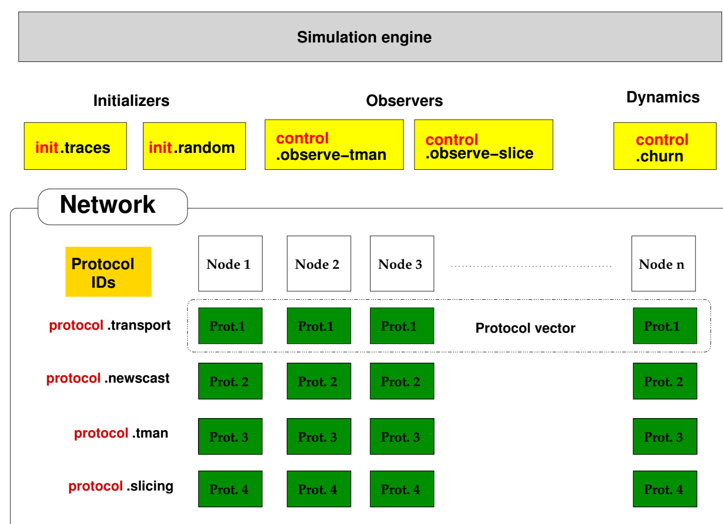
Philosophy

- Light core, small memory footprint, clean functionality
- Configurability
  - Every component can be dynamically loaded through the configuration file
  - (Almost) every component of Peersim can be replaced with alternative implementations

Simulator engines

- Single-threaded simulator
- Cycle-Driven (CD)
  - Quick and dirty: no messages, no transport, synchronized
  - Specialized for epidemic protocols
  - Tested up to 10^7 nodes
- Event-Driven (ED)
  - More realistic: message-based, realistic transports
  - Can be used for both epidemic and normal protocols
  - Can run cycle-driven protocols
  - Tested up to 2.5×10^5 nodes

Architecture



Main interfaces offered to the programmer

- `Node` – a container of protocols
- `Protocol` – implementation of a protocol
- `Linkable` – to access properties of nodes
- `Control` – observer and modifier of sim properties

The *Node* interface

```
public interface Node extends Fallible, Cloneable
{

    /**
     * Returns the i-th protocol in this node. If i is not a valid
     * protocol id (negative or larger than or equal to the number
     * of protocols), then it throws IndexOutOfBoundsException.
     */
    public Protocol getProtocol(int i);

    /** Returns the number of protocols included in this node. */
    public int protocolSize();

    /**
     * Returns the unique ID of the node. It is guaranteed that the
     * ID is unique during the entire simulation, that is, there will
     * be no different Node objects with the same ID in the system
     * during one invocation of the JVM.
     */
    public long getID();
}
```


The *Protocol* interface

```
public interface Protocol extends Cloneable
{

    /**
     * Returns a clone of the protocol. It is important to pay
     * attention to implement this carefully because in PeerSim
     * all nodes are generated by cloning except a prototype node.
     * That is, the constructor of protocols is used only to
     * construct the prototype. Initialization can be done
     * via {@link Control}s.
     */
    public Object clone();

}
```


CDProtocol interface

```
/**
 * Defines cycle driven protocols, that is, protocols that have a
 * periodic activity in regular time intervals.
 */
public interface CDProtocol extends Protocol
{

    /**
     * A protocol which is defined by performing an algorithm in more
     * or less regular periodic intervals. This method is called by
     * the simulator engine once in each cycle with the appropriate
     * parameters.
     *
     * @param node the node on which this component is run
     * @param pid  the id of this protocol in the protocol vector
     */
    public void nextCycle(Node node, int pid);
}
```

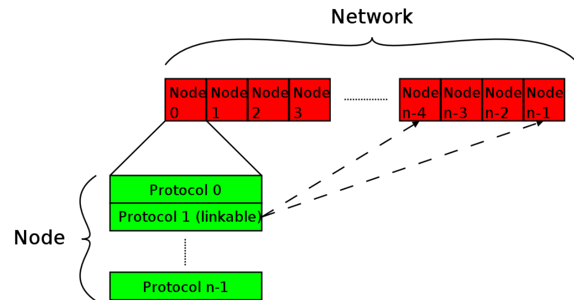

EDProtocol interface

```
/**
 * The interface to be implemented by protocols run under the
 * event-driven model. A single method is provided, to deliver
 * events to the protocol.
 */
public interface EDProtocol extends Protocol
{

    /**
     * This method is invoked by the scheduler to deliver events to
     * the protocol. Apart from the event object, information about
     * the node and the protocol identifier are also provided.
     * Additional information can be accessed through the
     * {@link CommonState} class.
     *
     * @param node the local node
     * @param pid the identifier of this protocol
     * @param event the delivered event
     */
    public void processEvent( Node node, int pid, Object event );

}
```

The *Linkable* interface



The *Linkable* interface

```
public interface Linkable extends Cleanable {
        /**
        * Returns the size of the neighbor list.
        */
        public int degree();

        /**
        * Returns the neighbor with the given index. The contract is that
        * listing the elements from index 0 to index degree()-1 should list
        * each element exactly once if this object is not modified in the
        * meantime. It throws an IndexOutOfBounds exception if i is negative
        * or larger than or equal to {@link #degree}.
        */
        public Node getNeighbor(int i);
```


The *Linkable* interface

```
/**
* Add a neighbor to the current set of neighbors. If neighbor
* is not yet a neighbor but it cannot be added from other reasons,
* this method should not return normally, that is, it must throw
* a runtime exception.
* @return true if the neighbor has been inserted; false if the
*     node is already a neighbor of this node
*/
public boolean addNeighbor(Node neighbour);

/**
* Returns true if the given node is a member of the neighbor set.
*/
public boolean contains(Node neighbor);
```


The *Control* interface

- Interface used to define operations that require global network knowledge and management, such as:
  - Initializers, executed at the beginning of the simulation
    - Initial topology
    - Nodes state
  - Dynamics, executed periodically during the simulation
    - Adding nodes
    - Removing nodes
    - Resetting nodes
  - Observers, executed periodically during the simulation
    - Aggregated values from all the nodes

The *Control* interface

```java
/**
 * Generic interface for classes that are responsible for
 * observing or modifying the ongoing simulation. It is designed
 * to allow maximal flexibility therefore poses virtually no
 * restrictions on the implementation.
 */
public interface Control
{

	/**
	 * Performs arbitrary modifications or reports arbitrary
	 * information over the components.
	 * @return true if the simulation has to be stopped, false
	 *   otherwise.
	 */
	public boolean execute();

}
```


CDSimulator

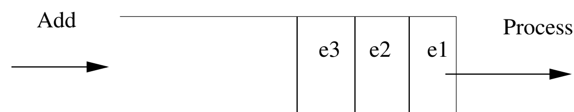
```
for i := 1 to simulation.experiments do
  create Network
  create prototype Node:
    for i:= 1 to #protocols do
          create protocol instance
  for j := 1 to network.size do
    clone prototype Node into Network
  create controls (initializers, dynamics, observers)

  execute initializers

  for k := 1 to simulation.cycles do
    for j := 1 to network.size do
      for p := 1 to #protocols
        execute Network.get(j).getProtocol(p).nextCycle()
    execute controls
    if (one control returned true) then
      break
```


EDSimulator

- Fully static singleton class that manages an event queue



EDSimulator

The engine executes `nextExperiment()`:

1. Run the initializers
2. Schedule the execution of all the controls in the event queue, according to their `Scheduler` config parameter

EDSimulator

```
for i := 1 to simulation.experiments do

    initialize MinHeap events

    create Network
    create prototype Node:
        for i:= 1 to #protocols do
            create protocol instance
    for j := 1 to network.size do
        clone prototype Node into Network
    create controls (initializers, dynamics, observers)

    execute initializers

    time = 0
    while (time < simulation.endtime) do
        (node, pid, e) = events.getMin();
        node.getProtocol(pid).processEvent(node, pid, event)
        if (event is a control that returned true) then
        break
```


EDSimulator

Configuration files

The configuration files specify

1. The protocols and the control to be executed, plus their order
2. The initializers, and their order
3. Unspecified items in the order lists are intended as to be executed in the order they've been declared

EDSimulator

Simultaneity

What happens when two (or more) events are scheduled to be executed at the same time?

- Control events: the order is defined by the config file
  - When undefined, it follows the order of declaration
- Non-control events: random order of execution

EDSimulator

Important engine's properties

- The engine offers a method `add(delay, event, node, pid)` to manually add events in queue
- The engine itself does not run protocols

Summarizing...

It just sends event messages. Protocols react to messages or perform cyclic actions.

EDSimulator

Important engine's properties

- At least one control or initializer must be declared which sends events to protocols. Why? They are *not* the only source of events
- Controls can be run at the end of the experiments (for example, to collect data)
- Controls returning `true` interrupt the experiment

Reasons to interrupt an experiment? Examples: Unexpected conditions (failure), some result has been found (success), ...

EDSimulator

The CDScheduler

A scheduler for periodic events. Why do we need this? E.g., to schedule periodic behaviours of gossip protocols

- Fills a gap between `CDSimulator` and `EDSimulator`
- Wraps cycles into events according to configuration files
- Fine-grained control (it works on a per-node basis)

What is the purpose of the config file?

- Once all the components have been implemented the whole simulation has to be set up
  - Declare what components to use
  - Define the way they should interact
- In Peersim simulations are defined through a plain text configuration file
- Configuration file is divided in 3 main parts
  - General setup
  - Protocol definition
  - Control definition

Configuration files

- Plain ASCII file containing key-value pairs
- Lines starting by # are ignored by the configuration reader

Syntax

{protocol,init,control}.string_id [full_path_]classname

- The class Initializer implements the interface Control
- An Initializer object is run at the beginning of the simulation

Configuration files

Component parameters' syntax

{protocol,init,control}.string_id.parameter_name parameter_value

- Previously declared

CDProtocol config

```
random.seed 1234567891

control.shf Shuffle

simulation.cycles 100
simulation.experiments 50

network.size 10^6
network.node peersim.core.GeneralNode
```


CDProtocol config

```
random.seed 1234567891

simulation.endtime 10^6
simulation.logtime 10^3

simulation.experiments 50

network.size 10^6
network.node peersim.core.GeneralNode
```


ED Protocols

```
MINDELAY 10   # Minimum delay is 10 ms
MAXDELAY 400  # Maximum delay is 400 ms
DROP 0.0      # Drop probability is 0

protocol.urt UniformRandomTransport
{
    mindelay MINDELAY
    maxdelay MAXDELAY
}

protocol.tr UnreliableTransport
{
    # Messages are actually delivered by urt
    transport urt

    # Here, we drop them with probability DROP
    drop DROP
}
# A simple implementation of Linkable that does nothing;
# just stores a set of neighbors
protocol.link IdleProtocol
```

ED Protocols

```
# The protocol defined in the previous slides
protocol.my GossipProtocol
{
    # Use the unreliable trasport defined above
    transport tr

    # Use the random topology defined above
    protocol  link

    # Use the default value for prob
}
```


Initializers and Controls

```
# Initialize link with 20 neighbors at random
init.rndgraph WireKOut
{
          k 20
          protocol link
}

# Traffic generator
control.traffic TrafficGenerator
{
          # Inject traffic on my protocol
          protocol my

          # Every 1000ms (1s)
          step 1000
}
```


Implementing an Event-Driven protocol: Class structure

```
public class GossipProtocol implements EDProtocol {

            // Parameters

            // Local variables

            // Constructor

            // processEvent (from EDProtocol)

            // other methods

}
```


Implementing an Event-Driven protocol: Parameters

```
/**
 * The identifier of the protocol that implements the
 * Linkable interface.
 * @config
 */
private static final String PAR_PROT = "protocol";

/**
 * The identifier of the protocol that implements the
 * Transport interface.
 * @config
 */
private static final String PAR_TRANSPORT = "transport";

/**
 * The probability of discarding messages already known
 * @config
 */
private static final String PAR_PROB = "p";
```

Implementing an Event-Driven protocol: Local variables

```
/** Container class for parameters */
class Parameters {
        /** Linkable */
        int pid;

        /** Transport */
        int tid;

        /** Probability to discard */
        int prob;
}

/** The local field that gives access to global parameters */
private Parameters p;

/** Other data structures depending on the protocol */
Map messages;

/** Counter: message sent */
int sent = 0;
```


Implementing an Event-Driven protocol: Constructor

```
public GossipProtocol(String prefix)
{
        // Read pids
        p = new Parameters();
        p.pid  = Configuration.getPid(prefix+ . +PAR_PROT);
        p.tid  = Configuration.getPid(prefix+ . +PAR_TRANSPORT);
        p.prob = Configuration.getDouble(prefix+ . +PAR_PROB, 0.5);

        // Initialize data structures
        set = new HashMap();
}

public Object clone() throws CloneNotSupportedException
{
        GossipProtocol prot = (GossipProtocol) super.clone();
        prot.p = this.p;

        // Initialize data structures
        set = new HashMap();
}
```


Implementing an Event-Driven protocol: Event handling

```
public void processEvent(Node mynode, int mypid, Object event)
{
        Boolean stop = map.get(event);
        if (stop == FALSE) {
                if (CommonState.r.nextDouble() < prob) {
                        stop = TRUE;
                        map.put(event, stop);
                }
        }
        if (stop == TRUE)
                return;
        else
                map.put(event, FALSE);
        Linkable l = (Linkable) mynode.getProtocol(p.pid);
        Transport t = (Transport) mynode.getProtocol(p.tid);
        if (l.degree() == 0)
                return;
        Node peer = l.getNeighbor(CommonState.r.nextInt(l.degree()));
        t.send(mynode, peer, event, mypid);
        sent++;
}
```


Implementing an Event-Driven protocol: Injecting load

```
public class TrafficGenerator implements Control {

        /** Parameter of the protocol where to inject messages*/
        private static final String PAR_PROT = "protocol";

        /** Protocol id */
        private final int pid;

        public TrafficGenerator(String prefix) {
                pid = Configuration.getPid(prefix+ . +PAR_PROT);
        }

        /** Generates one single message
          public boolean execute() {
          int size = Network.size();
          Node init = Network.get(CommonState.r.nextInt(size));
          EDSimulator.add(0, new Object(), init, pid);
          return false;
          }
}
```

Implementing an Event-Driven protocol: Measuring global behaviour

```
public class GossipObserver implements Control {

        /** Parameter of the protocol we want to observe */
        private static final String PAR_PROT = "protocol";

        /** Protocol id */
        private final int pid;

        /** Prefix to be printed in output */
        private final String prefix;

        public GossipObserver(String prefix) {
                this.prefix = prefix;
                pid = Configuration.getPid(prefix + "." + PAR_PROT);
        }
```


Implementing an Event-Driven protocol: Measuring global behaviour

```
        /** Analyze global network */
        public boolean execute() {
                int size = Network.size();
                Set set = new HashSet();
                for (int i=0; i < size; i++) {
                        GossipProtocol p =
                                (GossipProtocol) Network.get(i).getProtocol(pid);
                        set.add(p.getMessages()); // Not shown; returns rcv msgs
                }
                int total = set.size();
                IncrementalStats stats = new IncrementalStats();
                for (int i=0; i < size; i++) {
                        GossipProtocol p =
                                (GossipProtocol) Network.get(i).getProtocol(pid);
                        stats.add(p.getMessages().size());
                }
                System.out.println(prefix + "." + stats);
        }
}
```


The Vector package

What can peersim.vector do for you?

- Compute statistics on your (vector of) protocols
- Initialize your (vector of) of protocols
  - with values distributed uniformly at random
  - with values distributed linearly, or a peak
  - with values stored in a file
- Copy values from one vector of protocols to another
- Normalize values stored in one of your protocols

Example: the Vector observer

```
/** Returns the number of messages sent */
public int getSent() { return sent; }
```

```
protocol.my GossipProtocol
{
        transport mytransport
        protocol  newscast
}

control.check VectorObserver
{
        protocol my
        getter getSent
}
```

Example: the Vector observer

```
// Customize the probability of discarding nodes at each
// protocol instance; not really meaningful, just an example
public void  setProb(double prob) { this.prob = prob; }
```

```
protocol.my GossipProtocol
{
        transport mytransport
        protocol  newscast
}

init.ages UniformDistribution
{
        protocol gossip
        setter setProb
        min 0.5
        max 0.6
}
```