

# Simulating overlay networks with PeerSim

*Moreno Marzolla*

Dipartimento di Scienze dell'Informazione  
Università di Bologna

<http://www.moreno.marzolla.name/>

# Acknowledgements

- These slides are based on those kindly provided by Andrea Marcozzi
- Part of this material is taken from Alberto Montresor and Gianluca Ciccarelli presentation “PeerSim: Informal Introduction”

# Outline

- Introduction to Peersim
  - What is Peersim
  - Peersim components
- Aggregation
  - What is aggregation
  - Average in Peersim

# Introduction: P2P Systems

- Peer-to-Peer systems are potentially huge (composed of millions of nodes);
- Nodes join and leave the network continuously;
- Evaluating a new protocol in a real environment is not an easy task

# What is PeerSim?

- PeerSim is an open source P2P systems simulator developed at the Department of Computer Science, University of Bologna
- It has been developed with Java
- Available on Source Forge (*<http://peersim.sf.net>*)
- Its aim is to cope with P2P systems properties
- High Scalability (up to 1 million nodes)
- Highly configurable
- Architecture based on pluggable components

# What is a simulator?

- *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.*

# The peersim Simulation Engine

## ■ Cycle-Driven (CD)

- Quick and dirty: no messages, no transport, synchronized
- Specialized for epidemic protocols
- Tested up to  $10^7$  nodes

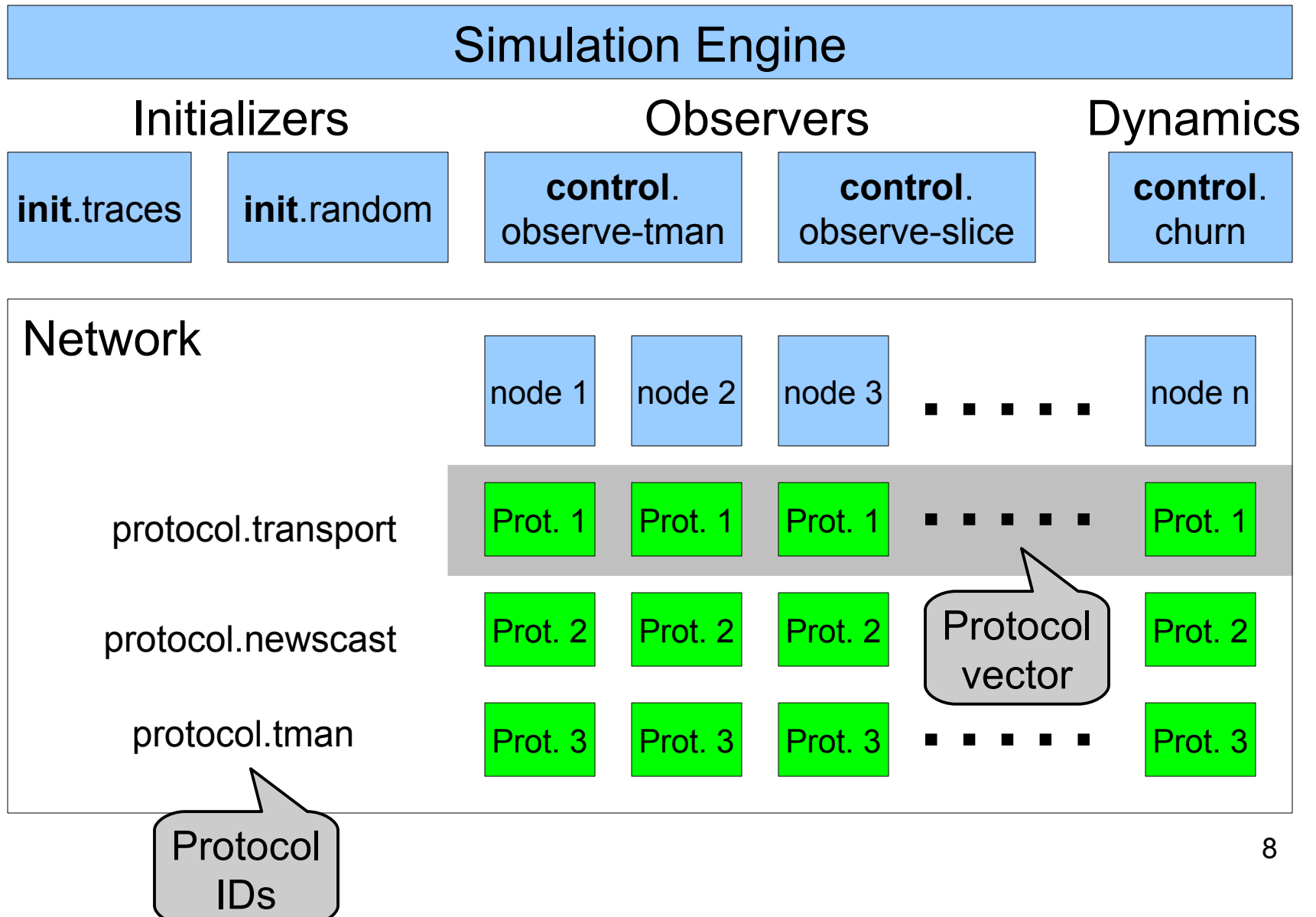


That's us

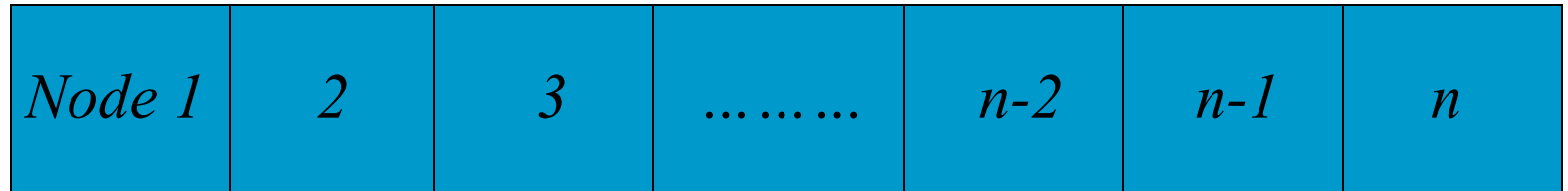
## ■ Event-Driven (ED)

- More realistic: message-based, realistic transports
- Can be used for both epidemic and normal protocols
- Can run cycle-driven protocols

# Peersim

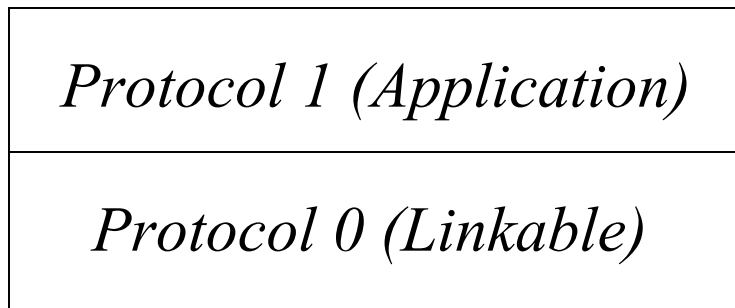


# Network Representation (I)



*Network*

*Node*



- A node is a stack of protocols
- `node.getProtocol(pid)`

# Network Representation (II)

- Network: global array which contains all the network nodes
- Node: each node's state and actions are described through a stack of protocols. Protocols are accessed through a *Pid*
- Linkable: interface used to access and manage node's view and other properties of a node

# Network Representation (III)

- CDProtocol: interface used to describe node's actions at each cycle.
  - A generic node can both perform local actions (CDProtocol) and manage the local view (Linkable)
- Control: performs the global initialization and performance analysis
  - Note: initializers are just Control objects with the peculiarity of being executed just once at the beginning

```

public interface Node extends Fallible, Cloneable
{
    /**
     * Returns the <code>i</code>-th protocol in this node. If <code>i</code>
     * 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.
     * Preferably nodes
     * should implement <code>hashCode()</code> based on this ID.
     */
    public long getID();

    /* ... */
}

```

```
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();

}
```

# Main Interfaces: Protocol

- The *CDProtocol* interface is used to define cycle-driven protocols, that is the actions performed by each *node* at each simulation cycle
- Each node can run more than one protocol
- Protocols are executed sequentially

```

/**
 * 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 protocolID
 *         the id of this protocol in the protocol array
 */
public void nextCycle(Node node, int protocolID);

}

```

```

/**
 * 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 );
}

```

# Main Interfaces: Linkable

- *Linkable* is used to manage node's view.  
Typical actions are:
  - Add neighbour
  - Get neighbour
  - Node's degree
  - Note: the Linkable interface does **not** allow to remove a neighbour; you need to define your own interface to do so

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

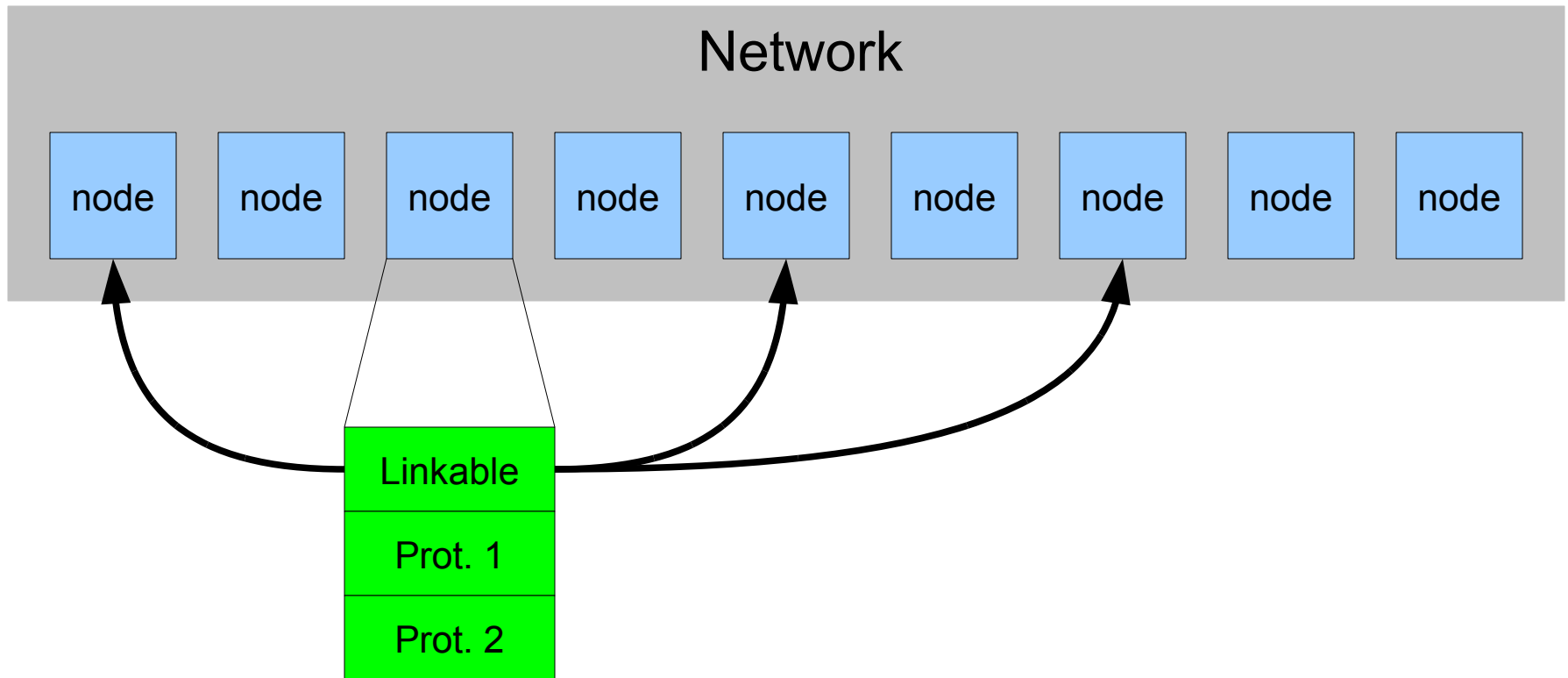
    /**
     * Returns the neighbor with the given index.
     */
    public Node getNeighbor(int i);

    /**
     * Add a neighbor to the current set of neighbors.
     */
    public boolean addNeighbor(Node neighbour);

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

    /**
     * A possibility for optimization. An implementation should try to
     * compress its internal representation. Normally this is called
     * by initializers or other components when
     * no increase in the expected size of the neighborhood can be
     * expected.
     */
    public void pack();
}
```

# The Linkable interface



# 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

# The Control interface

```
/**
 * 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
{

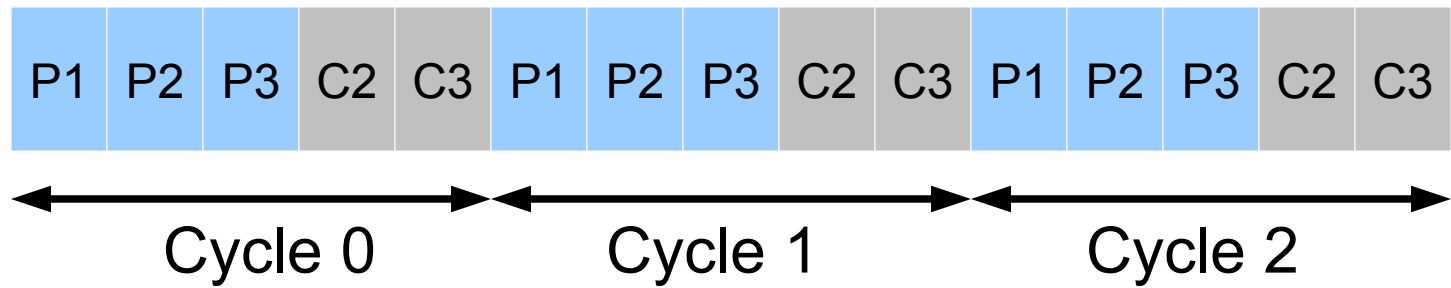
/**
 * Performs arbitrary modifications or reports arbitrary information over the
 * components.
 * @return true if the simulation has to be stopped, false otherwise.
 */
public boolean execute();

}
```

# CD Simulator

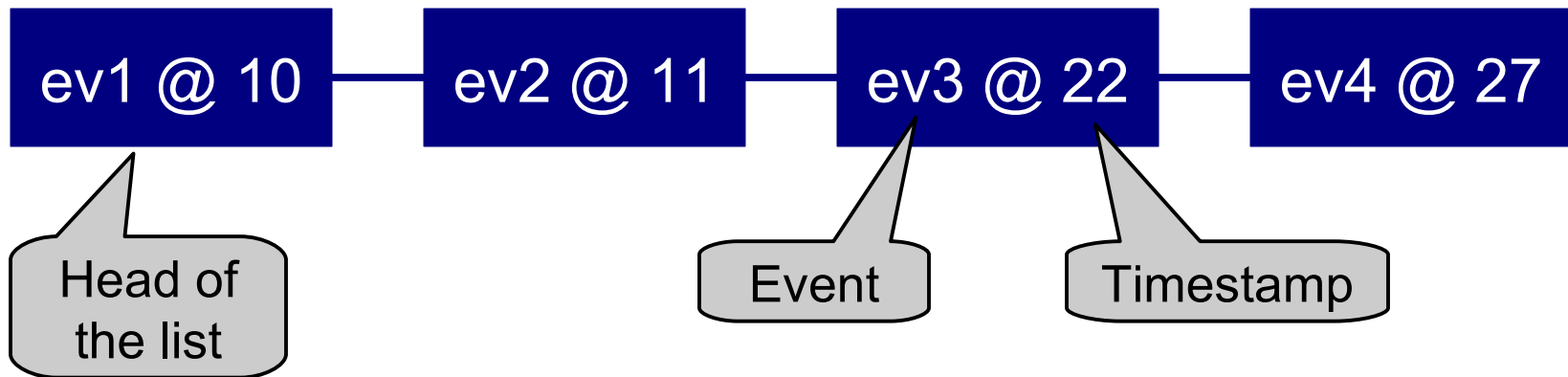
```
for i := 1 to simulation.experiments do
  create Network
  create prototype Node :
    for i := 1 to #protocols do
      create protocol instance
  for j := 1 to network.size do
    clone prototype Node into Network
  create controls ( initializers, dynamics, observers )
  execute initializers
  for k := 1 to simulation.cycles do
    for j := 1 to network.size do
      for p := 1 to #protocols do
        execute Network.get(j).getProtocol(p).nextCycle()
    execute controls
    if ( one control returned true ) then
      break
```

# CD Simulator



# EDSimulator

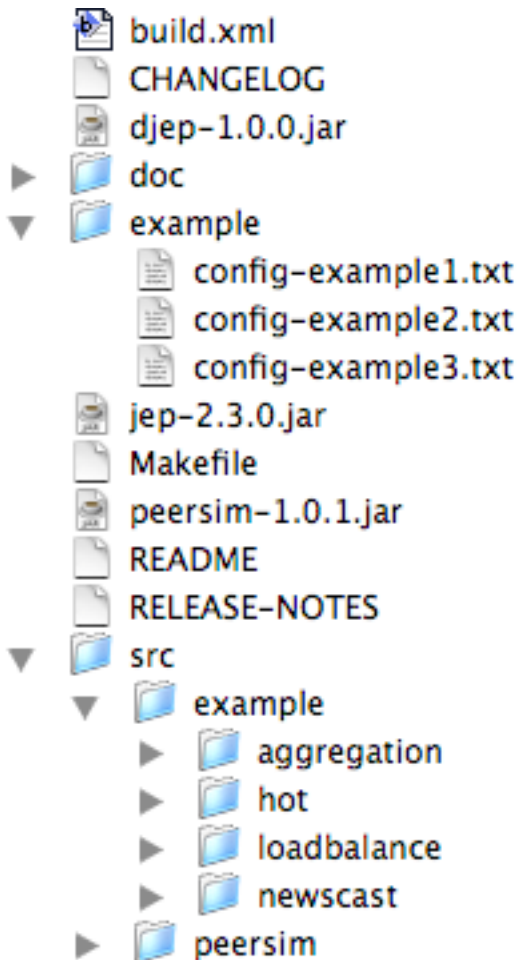
- In the Event-Driven paradigm, the simulator handles an *event queue*, which contains *events* with an associated *timestamp*



# 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
  createcontrols ( 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 acontrol that returned true) then
      break
```

# Peersim tree



- *SRC* contains the source code of Peersim and of some example protocols (aggregation, newscast)
- The parameters for the simulation are specified by a *Configuration File*

# Peersim Configuration

- 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

# Peersim Configuration

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

- Syntax:

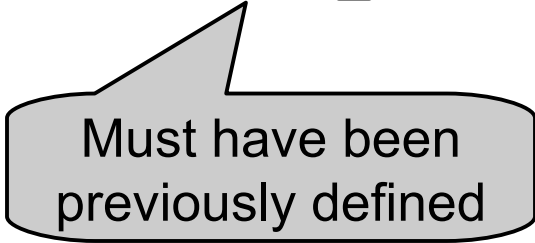
```
{protocol,init,control}.string_id  
[full_path]classname
```

- The class Initializer implements the interface Control
- An Initializer object is run at the beginning<sub>28</sub> of the simulation

# Peersim Configuration

- Component parameters' syntax

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



Must have been  
previously defined

# Example

```
random.seed 1234567891
```

Global property: used to initialize the RNG

```
control.shf Shuffle
```

Shuffles the order in which nodes are visited at each cycle

```
simulation.cycles 100
```

Maximum number of simulation cycles

```
simulation.experiments 50
```

```
network.size 10^6
```

Number of nodes in the network

```
network.node peersim.core.GeneralNode
```

# Aggregation

# Example: AverageFunction

**1.** *AverageFunction*

**0.** *Newscast*

By Aggregation we mean calculating a certain function (eg. average) over a number of values distributed over a network

# Example: AverageFunction (II)

- Each node has a numeric value
- The aggregate value has to be known by each node
- Local value of node  $i$  contains current estimate of the average
- Node  $i$  select random peer  $j$ :
  - $i$  and  $j$  exchange estimates
  - $i$  and  $j$  update local estimate with the average

# PeakDistributionInitializer

- All peers in the network hold a value
- We need to initialize the values such that
  - One peer (peer #0) holds a “large” value
  - The other peers maintain zero

# PeakDistributionInitializer

```
/**
 * Initialize an aggregation protocol using a peak distribution; only one peak
 * is allowed. Note that any protocol implementing
 * {@link peersim.vector.SingleValue} can be initialized by this component.
 */
public class PeakDistributionInitializer implements Control {

    // Constants

    // Fields

    // Constructor

    // Methods

}
```

# PeakDistributionInitializer

```
// -----  
// Constants  
// -----  
  
/**  
 * The load at the peak node.  
 *  
 * @config  
 */  
private static final String PAR_VALUE = "value";  
  
/**  
 * The protocol to operate on.  
 *  
 * @config  
 */  
private static final String PAR_PROT = "protocol";
```

# PeakDistributionInitializer

```
// -----  
// Fields  
// -----  
  
/** Value at the peak node.  
 * Obtained from config property {@link #PAR_VALUE}. */  
private final double value;  
  
/** Protocol identifier; obtained from config property {@link #PAR_PROT}. */  
private final int pid;  
  
// -----  
// Constructor  
// -----  
  
/**  
 * Creates a new instance and read parameters from the config file.  
 */  
public PeakDistributionInitializer(String prefix) {  
    value = Configuration.getDouble(prefix + "." + PAR_VALUE);  
    pid = Configuration.getPid(prefix + "." + PAR_PROT);  
}
```

# PeakDistributionInitializer

```
// -----  
// Methods  
// -----  
  
/**  
 * Initialize an aggregation protocol using a peak distribution.  
 * That is, one node will get the peak value, the others zero.  
 * @return always false  
 */  
public boolean execute() {  
    for (int i = 0; i < Network.size(); i++) {  
        SingleValue prot = (SingleValue) Network.get(i).getProtocol(pid);  
        prot.setValue(0);  
    }  
    SingleValue prot = (SingleValue) Network.get(0).getProtocol(pid);  
    prot.setValue(value);  
  
    return false;  
}
```

# AverageFunction

```
/**
 * This class provides an implementation for the averaging function in the
 * aggregation framework. When a pair of nodes interact, their values are
 * averaged. The class subclasses {@link SingleValueHolder} in
 * order to provide a consistent access to the averaging variable value.
 *
 * Note that this class does not override the clone method, because it does
 * not have any state other than what is inherited from
 * {@link SingleValueHolder}.
 */
public class AverageFunction extends SingleValueHolder implements CDProtocol
{
    /**
     * Creates a new {@link example.aggregation.AverageFunction} protocol
     * instance.
     *
     * @param prefix
     *         the component prefix declared in the configuration file.
     */
    public AverageFunction(String prefix) {
        super(prefix);
    }

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

# AverageFunction

```
/**
 * Using an underlying {@link Linkable} protocol choses a neighbor and
 * performs a variance reduction step.
 *
 * @param node
 *         the node on which this component is run.
 * @param protocolID
 *         the id of this protocol in the protocol array.
 */
public void nextCycle(Node node, int protocolID) {
    int linkableID = FastConfig.getLinkable(protocolID);
    Linkable linkable = (Linkable) node.getProtocol(linkableID);
    if (linkable.degree() > 0) {
        Node peer = linkable.getNeighbor(CommonState.r.nextInt(linkable
            .degree()));

        // Failure handling
        if (!peer.isUp())
            return;

        AverageFunction neighbor = (AverageFunction) peer
            .getProtocol(protocolID);
        double mean = (this.value + neighbor.value) / 2;
        this.value = mean;
        neighbor.value = mean;
    }
}
```

# AverageObserver

```
/**
 * Print statistics for an average aggregation computation. Statistics printed
 * are defined by {@link IncrementalStats#toString}
 */
public class AverageObserver implements Control {

    // Constants

    // Fields

    // Constructor

    // Methods

}
```

# AverageObserver

```
// //////////////////////////////////////  
// Constants  
// //////////////////////////////////////  
  
/**  
 * Config parameter that determines the accuracy for standard deviation  
 * before stopping the simulation. If not defined, a negative value is used  
 * which makes sure the observer does not stop the simulation  
 *  
 * @config  
 */  
private static final String PAR_ACCURACY = "accuracy";  
  
/**  
 * The protocol to operate on.  
 *  
 * @config  
 */  
private static final String PAR_PROT = "protocol";
```

# AverageObserver

```
// Fields

/**
 * The name of this observer in the configuration. Initialized by the
 * constructor parameter.
 */
private final String name;

/**
 * Accuracy for standard deviation used to stop the simulation; obtained
 * from config property {@link #PAR_ACCURACY}.
 */
private final double accuracy;

/** Protocol identifier; obtained from config property {@link #PAR_PROT}. */
private final int pid;

// Constructor

/**
 * Creates a new observer reading configuration parameters.
 */
public AverageObserver(String name) {
    this.name = name;
    accuracy = Configuration.getDouble(name + "." + PAR_ACCURACY, -1);
    pid = Configuration.getPid(name + "." + PAR_PROT);
}
```

# AverageObserver

```
/**
 * Print statistics for an average aggregation computation. Statistics
 * printed are defined by {@link IncrementalStats#toString}. The current
 * timestamp is also printed as a first field.
 *
 * @return if the standard deviation is less than the given
 *         {@value #PAR_ACCURACY}.
 */
public boolean execute() {
    long time = peersim.core.CommonState.getTime();

    IncrementalStats is = new IncrementalStats();

    for (int i = 0; i < Network.size(); i++) {
        SingleValue protocol = (SingleValue) Network.get(i)
            .getProtocol(pid);
        is.add(protocol.getValue());
    }

    /* Printing statistics */
    System.out.println(name + ": " + time + " " + is);

    /* Terminate if accuracy target is reached */
    return (is.getStD() <= accuracy);
}
```

# Peersim Scheduler

- Hence a Peersim simulation is scheduled in this way:

```
Initializers execution  
while (time < cycles) {  
    for each node in the Network {  
        CDProtocol actions are executed  
    }  
    Control actions are executed  
}
```

# Configuration File

```
01 #random.seed 1150540268549
02 simulation.cycles 300
03 control.shf Shuffle
04 network.size 50000
05
06 protocol.0 example.newscast.SimpleNewscast
07 protocol.0.cache 20
08 protocol.1 example.aggregation.AverageFunction
09 protocol.1.linkable 0
10 order.protocol 0 1
11
12 init.0 peersim.dynamics.WireKOut
13 init.0.protocol 0
14 init.0.k 20
15
16 init.1 example.aggregation.PeakDistributionInitializer
17 init.1.protocol 1
18 init.1.value 10000
19
20 control.ob0 aggregation.AverageObserver
21 control.ob0.protocol 1
```

*General settings*

*Protocols settings*

*Control settings*

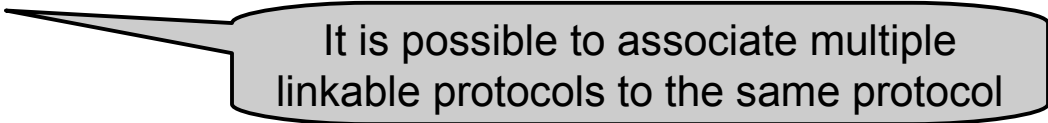
# Config File: *General Settings*

```
01 #random.seed 1150540268549  
02 simulation.cycles 300  
03 control.shf Shuffle  
04 network.size 50000
```

- Line 01: define the seed to be used. If not defined uses a random seed;
- Line 02: defines the number of cycles
- Line 03: if defined, shuffles the Network array at the beginning of each cycle
- Line 04: defines the Network size

# Config File: *Protocol Settings*

```
06 protocol.0 example.newscast.SimpleNewscast
07 protocol.0.cache 20
08 protocol.1 example.aggregation.AverageFunction
09 protocol.1.linkable 0
10 order.protocol 0 1
```



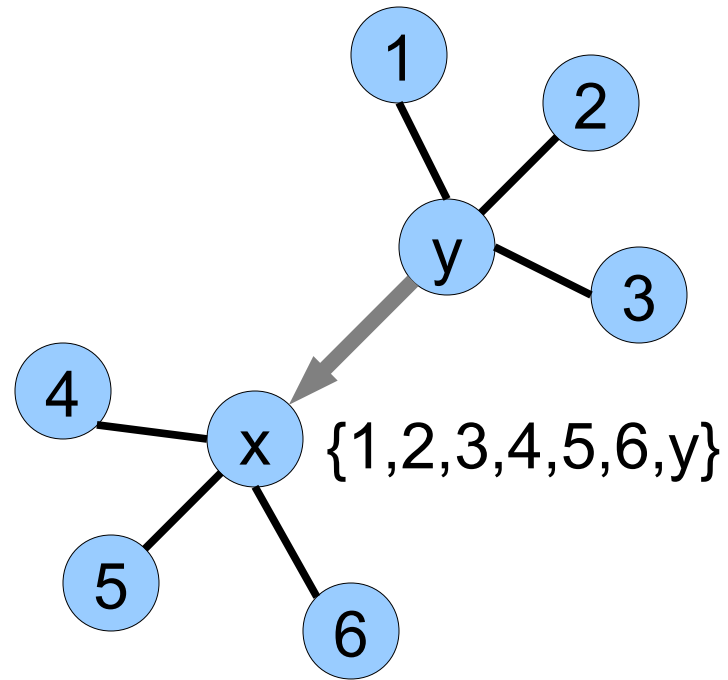
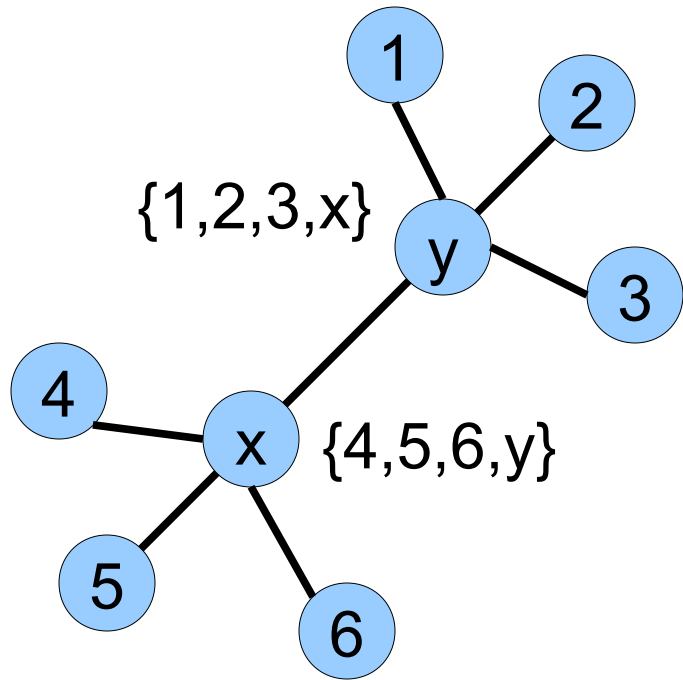
It is possible to associate multiple linkable protocols to the same protocol

- Line 06: define *protocol 0* as Newscast
- Line 07: define protocol 0's variable *cache* (it is the view size)
- Line 08: define *protocol 1* as the application calculating average aggregation
- Line 09: indicates that the linkable is Newscast (0)
- Line 10: indicated the order for protocols execution

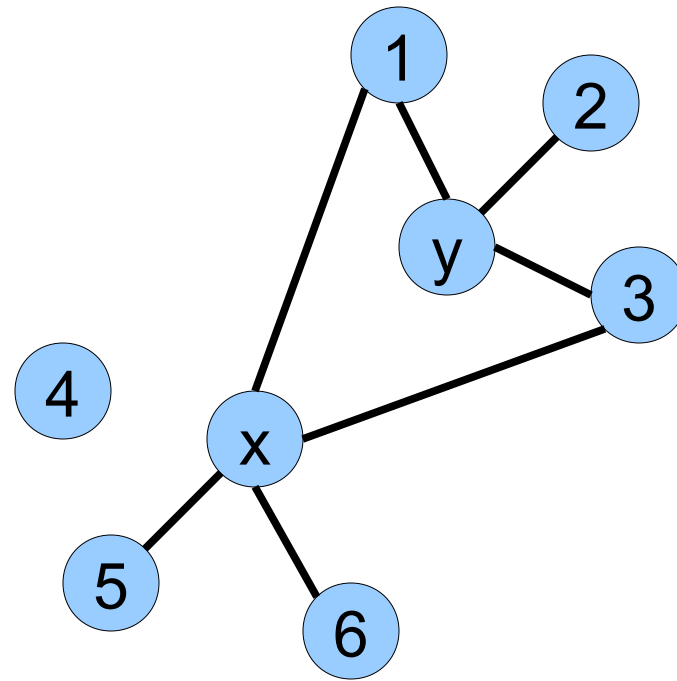
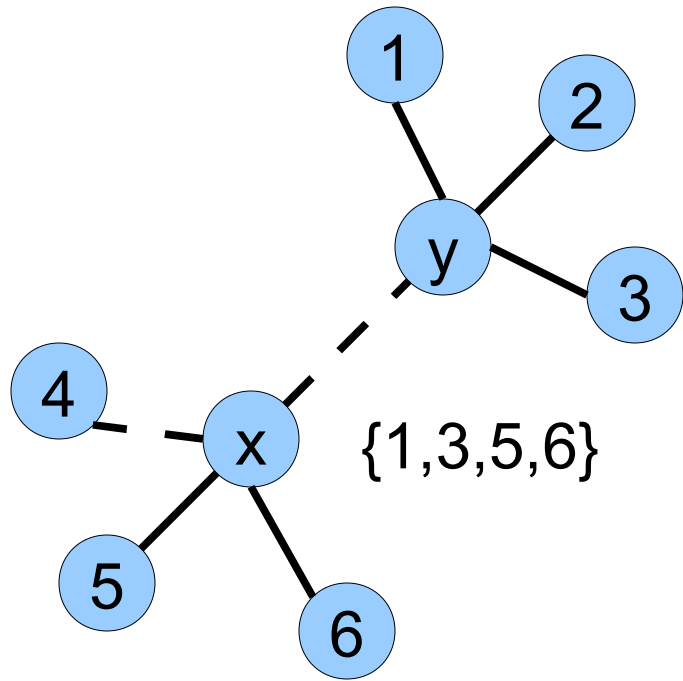
# Newscast

- Each node maintains a *local view*
  - Set of nodes it is connected to
- At each step, neighbors exchange their local views
- Each node merges the remote view with its local one, discarding old entries and keeping fresh ones

# Newscast



# Newscast



# Config File: *Control Settings*

```
12 init.0 peersim.dynamics.WireKOut
13 init.0.protocol 0
14 init.0.k 20
15
16 init.1 example.aggregation.PeakDistributionInitializer
17 init.1.protocol 1
18 init.1.value 10000
19
20 control.ob0 aggregation.AverageObserver
21 control.ob0.protocol 1
```

Build a random graph by wiring each node to k other randomly chosen nodes

- Lines 12 - 14: protocol 0 initialization
- Lines 16 - 18: protocol 1 initialization
- Lines 20 - 21: protocol 1 observer

# Aggregation: *execution*

```
CDSimulator: resetting
Network: no node defined, using GeneralNode
CDSimulator: running initializers
- Running initializer init.rnd: class peersim.dynamics.WireKOut
- Running initializer init.pk: class example.aggregation.PeakDistributionInitializer
CDSimulator: loaded controls [control.ao, control.shf]
CDSimulator: starting simulation
control.ao: 0 0.0 10000.0 50000 0.2 2000.0 49999 1
CDSimulator: cycle 0 done
control.ao: 1 0.0 5000.0 50000 0.2 999.9799999599992 49998 2
CDSimulator: cycle 1 done
control.ao: 2 0.0 2500.0 50000 0.2 223.5972844456889 49980 1
CDSimulator: cycle 2 done
control.ao: 3 0.0 1250.0 50000 0.2 95.67165454866102 49895 2
CDSimulator: cycle 3 done
control.ao: 4 0.0 625.0 50000 0.2 35.509880768063404 49472 3
CDSimulator: cycle 4 done
control.ao: 5 0.0 312.5 50000 0.2 13.630873692451484 47635 4
CDSimulator: cycle 5 done
control.ao: 6 0.0 156.25 50000 0.2 4.255885452309181 39859 2
CDSimulator: cycle 6 done
control.ao: 7 0.0 78.125 50000 0.2 1.3263675716346623 19357 2
CDSimulator: cycle 7 done
control.ao: 8 0.0 39.08306360244751 50000 0.2 0.4864481227847207 2380 2
CDSimulator: cycle 8 done
control.ao: 9 0.0 23.510593455284834 50000 0.19999999999999984 0.17639831008298432 22 1
CDSimulator: cycle 9 done
```

On the ReadMe file in the Peersim home directory there are the instructions for running the example

# Additional resources

- PeerSim Web Page  
<http://peersim.sf.net/>
- Class documentation  
<http://peersim.sourceforge.net/doc/index.html>
- Tutorial for the Cycle-based engine  
<http://peersim.sourceforge.net/tutorial1/tutorial1.pdf>