SICSIM: A Flow Level, Discrete Event Simulator for P2P Networks

Amir H. Payberah Fatemeh Rahimian
amir@sics.se fatemeh@sics.se

November 5, 2008

Contents

1 Introduction

SICSIM is a *Discrete Event*, *Flow Level* simulator for simulating Peer-to-Peer overlays. Although SICSIM does not consider the details of each packet delivery, it models some of the properties of the underlying network, such as the latency of network and bandwidth of links. What is more, it comes along with a streaming layer, which eases the implementation of Peer-to-Peer streaming overlays.

Discrete Event Simulation
Discrete event simulation is a widespread technique for computer network simulation. A discrete event simulator models a system whose state may change only at discrete points in time. In this model, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. In this approach, the simulation time is usually advanced from the time of the current event to the time of the next scheduled event simulation, i.e. it skips over periods of inactivity, which results in a less computation time.

Flow Level Simulation
A traditional model to simulate network traffic is packet level simulation. Packet level simulation employs packet by packet modeling of network activities. In this model for each packet departure or arrival one event will be generated. Considering the effect of any single packet makes this approach accurate, but heavy weighted. If the size of network grows, huge number of events will be generated that results in costly processing time and significant complexity. Therefore, packet level simulation can not scale well. Another model, which simplifies simulating network traffic is flow level model. In this model the underlying layers of network are abstracted away and the events are generated only when the rate of flows change. This abstraction, enables simulations at large scale, but at the cost of losing accuracy. This is because the effects of underlying layers are ignored.

Underlying Network Properties
To deal with inaccuracy of flow level simulation, SICSIM incorporates some of the effects of the underlying network, which may influence the performance of system. In other words, although SICSIM abstracts away the details of the underlying layers, it does not ignore all the properties of the network and models some of them, such as link latency and bandwidth.

2 SICSIM Structure

The main modules of SICSIM are as follow:

- **Peer:** A peer models a node of the network and can be of any customized type.

- **Link:** Each peer connects to the system through a link. The links are referred to as *physical link* in Figure 1.
- **Core:** Each message, which is sent from one peer to another, is passed through core network. The core modules is aimed to model Internet, while abstracting away from the underlying layers of the network.
- **Network:** Network contains all the peers in the system, no matter whether they have already joined the overlay or not.
- **Overlay:** It contains the peers who have joined the overlay.
- **Monitor:** Monitor is an auxiliary object that have a global view of peers in network and can monitor the behaviour of each peer and structure of the whole overlay.
- **Failure Detector:** For the sake of simplicity we assume there is one failure detector in the system. Each peer can register for the peers of its interest. Upon failure of a peer, all the peers who have registered for that peer, will be notified.

Figure 1 shows each of these modules and their relation.

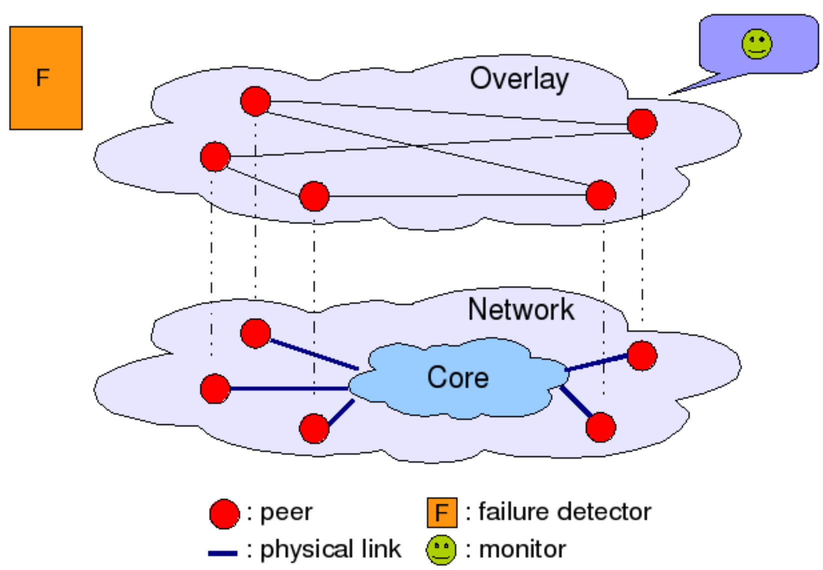

Figure 1: SICSIM main modules

3 Working With SICSIM

This section shows how to use SICSIM, and how to construct an overlay. More details of the source code are explained in the supplementary Javadoc [1].

3.1 Implementing The Overlay

To construct an overlay network, users should define two main modules: (i) the peers in the overlay and (ii) the links that connect the peers to the core network. There is already one type of link, called *reliable link*, implemented in SICSIM. Other type of links should be implemented by user, if required. In addition to peers and links, user could implement the monitor, if they want to have a global view to the system , e.g. to verify the overlay. However, defining a monitor is not mandatory.

The following explains the main classes and methods that users need to know to implement an overlay.

▷ `NodeId`
Each peer in an overlay is known by a `NodeId`. The `NodeId` has two parts, `id` and `ip` and it is represented as `id@ip`.

▷ `Message`
The messages that peers send to each other are instances of `Message` class. The `Message` has two fields: `type` that shows type of the message, and `data` that shows content of the message.

▷ `FailureDetector`
There is an instance of `FailureDetector`, called `failureDetector`, in the system. Peers can use this `failureDetector` to detect the failure of peers of their interest. The main methods of this class are `register` and `unregister`.

▷ `OverlayNetwork`
A peer is joined to the overlay, if it belongs to the `OverlayNetwork`. The main methods of this class are as follow:

- `add`: To add a peer to the overlay network.
- `remove`: To remove a currently joined peer from the overlay.
- `getRandomNodeIdFromNetwork`: To get the `NodeId` of a random node from the overlay. The method returns `null`, if it can not find any peer.

▷ `Bandwidth`
The peers who inherit from `BandwidthPeer` (see `Peer`), has access to an instance of `Bandwidth` class. The peers use this class to find out their current bandwidth usage, i.e. their upload and download rate.

[1] http://www.sics.se/∼amir/sicsim/javadoc

- `getTotalUploadBandwidth`: This method returns the total bandwidth usage of output link of a peer.
- `getTotalDownloadBandwidth`: This method returns the total bandwidth usage of input link of a peer.
- `getCurrentUploadRate`: This method shows the current rate of transferring data from one peer to another.

▷ `Peer`

The main entity of each overlay is its peers. SICSIM defines two types of peer: `AbstractPeer` and `BandwidthPeer`. The `AbstractPeer` contains the main methods that each peer requires. `BandwidthPeer` inherits from `AbstractPeer` and provides input and output bandwidth for the peers. These two classes are abstract classes and user should implement the required methods. Following are some of the main methods that should be implemented for constructing the overlays:

- `init`: This method can be used to initialize the peer.
- `create` and `join`: Whenever a new peer is created these methods are called. The difference is that, `create` is called for the first peer in the system and `join` is called for the rest of peers. These methods are defined as abstract methods and should be implemented by user.
- `receive`: This method is an abstract method, which is called whenever a peer receives a message from other peers.
- `failure`: This method is called by the failure detector to notify the peer of the failure of another peer. The user should implement the appropriate event handler in this method.
- `leave`: Whenever a peer wants to leave the system, the simulator calls this method. In this method, the user defines the steps that the peer should take before leaving the system. When the peer is done with this method, it should send a message to the simulator to acknowledge that it is ready to leave. More details about how to send a message to the simulator is explained in `sendSim`.
- `sendMsg`: This method is implemented in `AbstractPeer` and the users can use it to send control messages (message for short) to other peers. `sendMsg` has two arguments as input: `destId` is the address of destination peer and `msg` is the message sent to that peer.
- `startSendData` and `stopSendData`: There are two types of messages in SICSIM: (i) control messages and (ii) data messages. The control messages are considered to have a very small size, which would use zero bandwidth, and are sent by calling `sendMsg`. On the other hand, data messages are real data and can be of any type, e.g. text, video, audio, images, etc.

For the sake of simplicity, we do not transfer real data in the simulator. We just define the start and stop of the stream. We assume there is a flow of data from one peer to another with a certain rate. The beginning and the end of sending data messages is acknowledged by `startSendData` and `stopSendData`, respectively. These two methods are implemented in `BandwidthPeer` and users can use them.

- `loopback`: Each peer can use this method to send a message (a control message) to itself.
- `sendSim`: This method sends a message directly to the simulator. For example if a peer wants to leave completely the system, it should send a message to the simulator and asks it to remove it from system, i.e. `sendMsg(new Message("LEAVE_GRANTED", null))`.
- `broadcast`: This method sends a message to all peers who have joined the overlay network.
- `signal`: If a user wants to send some commands to the peers, he/she should define a signal scenario (see 3.3). By defining signal scenario the simulator sends the signals to a number of peers, by calling `signal` method of selected peers. In this methods the user should define the behaviour of peers by receiving that signal.
- `syncMethod`: If in one case a user wants that all peers in the system do one specific task in each time unit, he/she should enable this feature in SICSIM, by setting `true` value for `SYNC_UPDATE` the in configuration file. If this feature is enabled, the simulator calls `syncMethod` of all peers at each time unit.

▷ `Link`
The peers are connected to the system through the links. The abstract class of links is `AbstractLink`. To create a new type of link, a class, which inherits from `AbstractLink`, should be implemented. There is already one implemented and ready-to-use type of link in SICSSIM, i.e. `ReliableLink`. A reliable link models a link that eventually delivers any message, which is sent through it.

▷ `Monitor`
Monitor is an abstract class that can help users to have a global view of peers in system and the overlay network. Having monitor is not mandatory and it can be disabled in the configuration file (by setting `Monitor` variable). The following lists the main methods of `Monitor`, which should be implemented by user:

- `update`: If `Monitor` is enabled, this method is called in each time unit. It can get the status of peers in that time unit or send an order to them.
- `snapshot`: This method is called periodically with period `SNAPSHOT_PERIOD` (defined in the configuration file) and can be useful to get snapshot of the status of all peers.

- `verify`: At the end of simulation, this method can be used to verify the structure of the overlay.

3.2 Configuring SICSIM

There are a few variables to configure SICSIM. These variables, which are listed below, are defined in a text file, named `sicsim.conf`.

- `SIM_TIME`: Defines the maximum time of simulation.
- `MAX_NODE`: Defines the ID space or maximum number of peers in the system.
- `SEED`: Defines the seed for random number generation.
- `SYNC_UPDATE`: Enables/disables the synchronous update feature of simulator. If it is set to `true` the simulator calls the `syncMethod` of all peers in system at every time unit.
- `MONITOR`: Enables/disables the `Monitor`.
- `SNAPSHOT_PERIOD`: Defines the period for getting snapshot from peers, by calling the `snapshot` method of `Monitor`.
- `NETWORK_LATENCY`: Defines the mean latency of the core network.
- `NETWORK_LATENCY_DRIFT`: Defines the maximum latency drift of the core network (To have a more realistic model of latency, SICSIM assumes the latency between each two peers in not fixed and changes over time).
- `LINK_LATENCY`: Defines the maximum link latency of a link.
- `FAILURE_DETECTOR_LATENCY`: Defines the upper bound latency for failure detection, i.e. the maximum latency after a node is failed and before a peer is notified of this event by the failure detector.
- `SKEWED`: The peer IDs in ID space can be distributed either uniformly or skewed, if this variable is set to `false` or `true`, respectively.
- `NUM_OF_CLUSTER`: Shows the number of clusters in the ID space, for a skewed distribution.
- `PROB_OF_CLUSTER`: Shows the percentage of peers that are places in clusters of a skewed distribution. For example if `NUM_OF_CLUSTER` is 4 and `PROB_OF_CLUSTER` is 0.7, means there are 4 clusters in ID space and 70% of whole peers are placed in these clusters.
- `LOG_SIM`: Enables/disables the printing out the debugging information of simulator core.

- `LOG_LEVEL`: Shows the level of printing/debugging information. 0 means *disable*, 1 means *error*, 2 means *warning*, 3 means *notice*, 4 means *info* and 5 means *debug*. If it sets to 5, every debugging messages of SICSIM core is printed, but if it sets to 3, only notice messages are printed, and setting it to 0, disable the printing debug information.
- `SCENARIO_FILE`: Defines the name of scenario file.
- `NET_SIZE_FILE`: Defines the name of the file used to store size of the network over time.
- `BW_FILE`: Defines the file name to store the bandwidth status of peers.
- `NETWORK_FILE`: Defines the file name to store the information of peers in the overlay network.
- `FEL_FILE`: Defines the file name to store future event list status.
- `TIME_FILE`: Defines the file name to store the time of saving system status.
- `OVERLAY_FILE`: Defines the file name to store the overlay information.
- `FAILURE_DETECTOR_FILE`: Defines the file name to store the status of failure detector.
- `BUFFER_SIZE`: Defines the buffer size used in peer of type `BandwidthPeer`.
- `NUM_OF_STRIPES`: In case of using SICSIM for simulating media streaming systems, if the media stream is split into a number of stripes, this variable defines the number of stripes.
- `STRIPE_RATE`: This variable defines the rate of each stripe, as a supplement to the previous variable.

3.3 Defining Scenario

The input scenario of the simulation is defined in a text file, with a name specified in `sicsim.conf`. The scenario file can have one scenario or multiple scenarios. Each scenario is separated by `"---"`. The last scenario does not need this separator at the end. Following are the different types of scenarios:

▷ **Lottery Scenario**
Lottery scenario defines the rate of joins, leaves and failures, as well as the interval between these events. Type of the peers and links are also defined. Here is an example:

```
type:      lottery
peer:      sicsim.samples.helloworld.Peer
link:      sicsim.network.links.ReliableLink
count:     100
interval:  20
join:      7
leave:     2
failure:   1
```

This scenario shows that the type of peers in system are `Peer`, and these peers are connected by the links of type of `ReliableLink` to the core network. It also asks the simulator to generate 100 events, of which 70% are joins, 20% are leaves and 10% are failures (from each 10 events (7+2+1), 7 are joins, 2 are leaves and 1 is failure). The inter-arrival time of the events is a exponential distribution with mean 20.

▷ **Delay Scenario**
It defines the delay between two consecutive scenarios. For example the following example produces a 1000 time unit delay, i.e. no new scenario is initiated for 1000 time unit.

```
type:   delay
delay:  1000
```

▷ **Monitor Scenario**
This type of scenario introduces a monitor into the system.

```
type:     monitor
monitor:  sicsim.samples.helloworld.OverlayMonitor
```

This would informs the simulator to create an instance of `OverlayMonitor` as a monitor of the overlay. Note that, such a scenario should be the first scenario, defined in the scenario file.

▷ **Signal Scenario**
If it is required to send some command to peers, the signal scenario can be used.

```
type:      signal
count:     12
interval:  10
signal:    5
```

The above scenario will send signal number 5 to 12 random peers in the network. The interval between sending these signal is exponentially distributed with mean 10.

▷ **Save Scenario**
This scenario can be used to save the state of the simulation.

```
type: save
```

▷ **Load Scenario**

By using this scenario, one can restore the previously saved overlay.

```
type: load
```

3.4 Some Useful Classes and Methods

▷ **Package sicsim.utils**

This package contains some classes, which may come handy for the simulation. More details about the methods of these classes are in Javadoc generated document.

- `Distribution`: This class contains the implementation of some important distribution, e.g. exponential, poisson, normal and etc.
- `FileIO`: This class contains some helpful methods for working with files. It has three main methods, `write` to write in a file, `append` to append at the end of a file, and `read` to read the file and return its content as a string.
- `MathMisc`: This class provides useful methods for working with IDs over a ring. For example it contains four different methods to check if one ID is placed in one interval or not. More clearly, method `belongsToI(long id, long start, long end, long n)` checks to see whether `id` is placed in `[start, end)` or not
- `PatternMatching`: This class contains some methods that helps to parse a file. In general, its methods have two arguments as input, first is the key that user is looking for its value, and second is the string that search is done over it. For example, `getNodeValue(str, "base_node")` searches in `str` and finds the value of `base_node` and returns it in `"id@ip"` format.

▷ **Save and Load Overlay**

In SICSIM users can save the state of the whole simulation at one point, and restore it later and continue from that point on. Save and load scenarios should be defines in the scenario file. Saving the overlay includes saving the status of the whole system, except the peers. In fact, user should define which properties of a peer should be saved and how it should be restored. For save, the user should overwrite the `toString` method of the peer, and for load the `restore` method of the peer should be implemented. Section 4.1 shows a sample of how to implement these two methods to save and load the status of peers.

4 Sample - Hello World

This section shows a simple overlay implemented in SICSIM. In this sample a peer of type `Peer`, inherited from `AbstractPeer`, is used as a main object of the overlay. Each peer connects to the core network through a `ReliableLink`. Whenever a peer comes to the system, it picks a random peer from the overlay and sends a `HELLO` message to it and joins the overlay.

When the destination peer receives the message, prints it out and adds the sender of the message to its friends and registers for it in the failure detector. This goes on periodically until the simulation time is over. Each `Peer` has two local variables: `friends`, which maintains the list of peers that have sent a hello message to the peer, and `failedFriends`, which maintains the list of those friends that have failed.

When a peer wants to leave the overlay, it first broadcasts a `LEAVE` message to all peers in the overlay and informs them that it is leaving and then leaves the system.

Moreover, the peers handle two types of signal, which are specified in the scenario file. Whenever a peer receives a signal, it detects the type of the signal and then picks a random peer from the overlay and sends it a `SIGNAL` message that carries the name of received signal as its data.

What is more, a subclass of `Monitor`, called `OverlayMonitor`, is used in this sample. The `OverlayMonitor` knows about the `friends` and `failedFriends` of all peers, by taking snapshots periodically. It also verifies the overlay at the end of the simulation.

Following explains more details about the structure of `Peer` and `OverlayMonitor`.

4.1 Peer

Procedure 1 shows the `create` method, which is called when the first peer joins the system. In line 2 this peer adds itself to the overlay and in line 3, it sends a `PERIODIC` message to itself by calling `loopback` method. This message is handled after `Peer.PERIOD_INTERVAL` time unit. Procedure 2 shows the `join` method. This

Procedure 1 Creating The Overlay

```
1: public void create(long currentTime) {
2:   this.overlay.add(this.nodeId);
3:   this.loopback(new Message("PERIODIC", null), Peer.PERIOD_INTERVAL);
4: }
```

method is called for each peer, who joins the system, except the first one. In this method, the peer first finds a random node in the overlay (line 2). If it finds a random node, it sends a `HELLO` message to that node (line 4). In line 5, it sends a `PERIODIC` message to itself by `loopback` method in order to find new nodes periodically. In line 6, the peer adds itself to the overlay.

When a peer wants to leave the system, it informs all the peers in the overlay about its leaving. As it is shown in procedure 3, the leaving peer broadcasts a

Procedure 2 Joining The Overlay

```
public void join(long currentTime) {
  NodeId randomNode = this.overlay.getRandomNodeIdFromNetwork();
  if (randomNode != null)
    this.sendMsg(randomNode, new Message("HELLO", ...));
  this.loopback(new Message("PERIODIC", null), Peer.PERIOD_INTERVAL);
  this.overlay.add(this.nodeId);
}
```

`LEAVE` message to all the peers (line 2). Then it asks the simulator to remove it from the system. It sends this request by calling `sendSim` method and sending a `LEAVE_GRANTED` message (line 3).

Procedure 3 Leaving The Overlay

```
1: public void leave(long currentTime) {
2:   this.broadcast(new Message("LEAVE", null));
3:   this.sendSim(new Message("LEAVE_GRANTED", null));
4: }
```

Procedure 4 shows how a peer reacts when it is informed of another peer's failure.

Procedure 4 Handling A Failure

```
public void failure(NodeId failedId, long currentTime) {
  if (!this.failedFriends.contains(failedId.toString()))
    this.failedFriends.addElement(failedId.toString());
}
```

Procedure 5 shows how a peer reacts to the receipt of a `SIGNAL` message from the simulator. Assume there are two types of signal in the system, known as 1 and 2. Whenever a peer receives a signal it sends `SIGNAL` message to a random peer with the signal number as its data.

When a peer receives a message from another peer Procedure 6 is called. In this procedure the proper handler is called according to the type of message. Procedure 7 shows how the event handlers can be registered. This procedure registers `handlePeriodicEvent` (Procedure 9) to handle `PERIODIC` messages, `handleHelloEvent` (Procedure 8) to handle `HELLO` messages, `handleSignalEvent` (Procedure 10) to handle `SIGNAL` messages, and `handleLeaveEvent` (Procedure 11) to handle `LEAVE` messages.

Procedure 5 Receiving A Signal From Simulator

```
public void signal(int signal, long currentTime) {
 String data1 = new String("Signal 1 from " + this.nodeId);
 String data2 = new String("Signal 2 from " + this.nodeId);
 NodeId randomNode = this.overlay.getRandomNodeIdFromNetwork(this.nodeId);
 if (randomNode == null)
  return;
 switch (signal){
  case 1:
   this.sendMsg(randomNode, new Message("SIGNAL", data1));
   break;
  case 2:
   this.sendMsg(randomNode, new Message("SIGNAL", data2));
   break;
  default:
   System.out.println("unknown signal number");
 }
}
```

Procedure 6 Receiving A Message

```
public void receive(NodeId srcId, Message data, long currentTime) {
 if (this.listeners.containsKey(data.type))
  this.listeners.get(data.type).receivedEvent(srcId, data);
 else
  System.out.println("This event is not registered!");
}
```

Procedure 7 Registering The Event Handlers

```
public void registerEvents() {
 this.addEventListener("PERIODIC", new PeerEventListener() {
  public void receivedEvent(NodeId srcId, Message msg) {
   handlePeriodicEvent();
  }
 });

 this.addEventListener("HELLO", new PeerEventListener() {
  public void receivedEvent(NodeId srcId, Message msg) {
   handleHelloEvent(srcId, msg);
  }
 });

 this.addEventListener("SIGNAL", new PeerEventListener() {
  public void receivedEvent(NodeId srcId, Message msg) {
   handleSignalEvent(srcId, msg);
  }
 });

 this.addEventListener("LEAVE", new PeerEventListener() {
  public void receivedEvent(NodeId srcId, Message msg) {
   handleLeaveEvent(srcId);
  }
 });
}
```

Procedure 8 Handling the HELLO Message

```
private void handleHelloEvent(NodeId srcId, Message msg) {
 System.out.println(this.nodeId + " receives message:  " + msg.data);
 if (!this.friends.contains(srcId.toString())) {
  this.friends.addElement(srcId.toString());
  this.failureDetector.register(srcId, this.nodeId);
 }
}
```

Procedure 9 Handling the PERIODIC Message

```
private void handlePeriodicEvent() {
 NodeId randomNode = this.overlay.getRandomNodeIdFromNetwork(this.nodeId);
 if (randomNode != null)
  this.sendMsg(randomNode, new Message("HELLO", ...));
 this.loopback(new Message("PERIODIC", null), Peer.PERIOD_INTERVAL);
}
```

Procedure 10 Handling the SIGNAL Message

```
private void handleSignalEvent(NodeId srcId, Message msg) {
  System.out.println(this.nodeId + " receives message:  " + msg.data);
}
```

Procedure 11 Handling the LEAVE Message

```
private void handleLeaveEvent(NodeId srcId) {
  if (this.friends.contains(srcId.toString())) {
    this.friends.removeElement(srcId.toString());
    this.failureDetector.unregister(srcId, this.nodeId);
    }
}
```

Procedures 12 and 13 show how to define `toString` and `restore` to save and restore the status of peers. The simulator save the state of the overlay whenever it reaches a `save` scenario in the scenario file. Likewise, it restore the values when it reaches a `load` scenario. The important thing here is to define `toString` method of peer. In this method the user should store the important variables of peers, e.g. `friends` and `failedFriends` in this sample. In addition to the local variables that the user defines in `Peer`, there are some other variables in peer's super class, i.e. `AbstractPeer` or `BandwidthPeer` that should be saved. To do this, the user should get the status of the super class by calling the `getStateString` method. This method returns a string that contains the important variable of the peer's super class that should be saved. The local variables of the peer should be added at the end of the returned string. The `getStateString` has two important arguments, the first one is the class of peer, and the second on is the class of link that connects the peers to the core network, e.g. `Peer.class` and `ReliableLink.class` in this sample.

Procedure 12 Saving the Currant Status of Peer

```
public String toString() {
  String str = this.getStateString(Peer.class, ReliableLink.class);
  Iterator<String> friendsIter = this.friends.iterator();
  str += "friends:  ";
  while (friendsIter.hasNext())
    str += friendsIter.next() + ", ";
  str += "\nfailed:  ";
  Iterator<String> failedIter = this.failedFriends.iterator();
  while (failedIter.hasNext())
    str += failedIter.next() + ", ";
  str += "\n";
  return str;
}
```

Procedure 13 Restore the Status of Peer

```
public void restore(String str) {
 String friendsList = PatternMatching.getStrValue(str, "friends:");
 String friendParts[] = friendsList.split(",");
 for (int i = 0; i < friendParts.length; i++)
  this.friends.addElement(friendParts[i]);
 String failedList = PatternMatching.getStrValue(str, "failed:");
 String failedParts[] = failedList.split(",");
 for (int i = 0; i < failedParts.length; i++)
  this.failedFriends.addElement(failedParts[i]);
}
```

As the `Peer` implements the abstract class of `AbstractPeer` it should implement all abstract methods of `AbstractPeer`. So in this sample the `Peer` contains the `syncMethod` method, but the body of this method is empty.

```
### Scenario 1 ###
type:               monitor
monitor:            sicsim.samples.helloworld.OverlayMonitor
---
### Scenario 2 ###
type:               lottery
peer:               sicsim.samples.helloworld.Peer
link:               sicsim.network.links.ReliableLink
count:              20
interval:           10
join:               4
leave:              0
failure:            1
---
### Scenario 3 ###
type:               delay
delay:              500
---
### Scenario 4 ###
type:               lottery
peer:               sicsim.samples.helloworld.Peer
link:               sicsim.network.links.ReliableLink
count:              2
interval:           10
join:               0
leave:              1
failure:            0
---
### Scenario 5 ###
type:               signal
count:              15
interval:           10
signal:             1
---
### Scenario 6 ###
type:               signal
count:              5
interval:           10
signal:             2
```

Scenario 1. Sample Scenario

Scenario 1 shows a sample scenario file for the hello world example. This scenario file contains 6 scenarios. Scenario 1 asks the simulator to create an instance of `OverlayMonitor` as the monitor. Scenario 2 is a `lottery` scenario. It asks the simulator to generate 20 events with the mean interval of 10 time units, and with the ratio of 4, 0 and 1 for `join`, `leave` and `failure`, respectively. Scenario 3 is a `delay` scenario that generates 500 time units delay after the previous scenario. Scenario 4 defines another `lottery` scenario, in which two peers leave the system. Scenario 5, which is a `signal` scenario, asks the simulator to send signal 1 to 15 random peers in the system. The mean interval for generating these signal events is set to 10. Finally, Scenario 6 sends signal 2 to 5 random peers in the system.

Note1: There should be at least one space between key and values in the configuration and scenario files.

Note2: Run the simulator as `"java sicsim.main.Main"`.

4.2 Monitor

`OverlayMonitor` has three main methods: (i) `update`, which is called at every time unit and prints out the current clock of simulator, (ii) `snapshot`, which gets a snapshot of the current state of peers, e.g. the list of `friends` and `failedFriends` of all peers and saves this snapshot in a file, and (iii) `verify`, which is called when the simulation is done with all the scenarios. It prints out the final state of peers, e.g. the final value of `friends` and `failedFriends`.