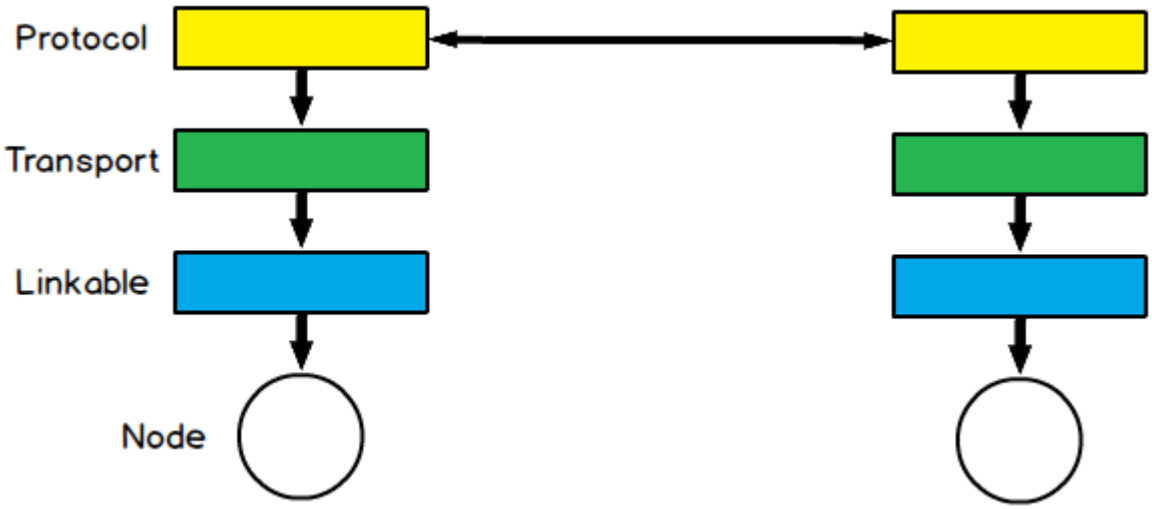
Peersim - How to use it

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 is not an easy task. 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.

The obvious advantage of using Peersim is the need to learn to program only two types of components (of Java classes) and for everything else to rely on an already functioning system, of which you just need understand the mechanics, rather than having to write everything from scratch.

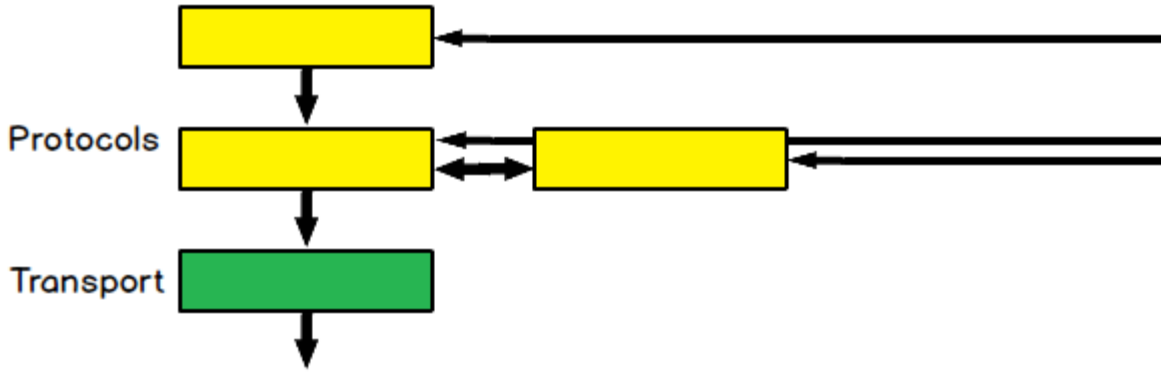
This tutorial aims to provide useful information to design and implement customized simulations, based on the user's requirements, thus saving the need to read the source code of the program, which should otherwise be a necessary task.
In particular, it will be necessary to understand how to write the classes that are useful for creating a customized protocol, without having to fully understand all the components.
A descriptive approach will be used without showing specific examples, but rather will illustrate in general the dynamics of operation of the simulator as a whole.



internal components of the nodes

For almost all customized simulations, it will be sufficient to learn to write only two types of components: *Protocol*, classes that implement the *Protocol* interface, and the *Control*, classes that implement the *Control* interface.
The first type clearly deals with protocols, the ones on which you want to experiment and the auxiliary ones to them, while the second one is made out of supervisors that observe the protocols and/or act by modifying the state of the latter.

The fundamental difference lies in the way in which they see the network; The *Control* see all the nodes and the network as a whole, so they have access to all of the information of the latter; the *Protocol* are contained in each node, they can act on the node itself and on the neighbour nodes that they can view.



various levels of Protocol

In addition *Protocol*, within the single node, can be arranged on several levels, forming a stack, in a completely analogous way to that of the ISO/OSI or TCP/IP representation.
There is also the possibility of putting multiple protocols in the same level.
Each instance of a protocol contained in a node will have access to the data and resources of the instances of the other protocols of a lower or equal level, contained in the node itself, and of the protocol's instances which are present in the other nodes.

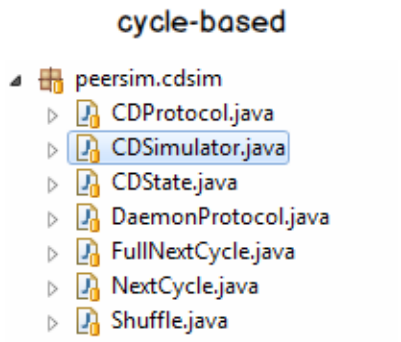
It is very useful to read the existing protocols and controls, firstly, to observe the best practices used in the writing of the latter, secondly, to avoid having to write from scratch classes that could be obtained by extending or taking a cue from existing ones.

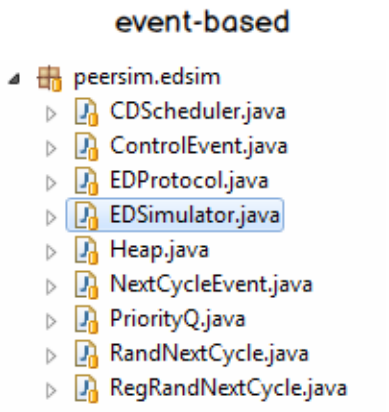
Both types of components have two important characteristics.
The first, fundamental, is the periodicity with which they are activated, a different one can be assigned to each component.
The second concerns with which components they are connected.
It implies to which data and resources of other components they have access, so what they can modify and what they can exploit. Remember that the link is unidirectional.

The ability to manage quickly and simply these two characteristics, in code terms, is one of Peersim's strengths, and highlights its versatility.
Another important thing to know is that the instances of the components can not be generated by the user in his code otherwise the simulator will not manage the functioning.

To observe the progress of the simulation, specific *Control* are used for this purpose, which therefore have the observer's characteristic. They can be used to print data even on files.

PeerSim supports two simulation models: the event-based model and the cycle-based model. The latter is simplified, which makes it possible to achieve extreme scalability and performance, at the cost of some loss of realism. However many simple protocols can sustain this loss without any problems. The simplified hypotheses of the cycle-based model are the lack of simulation of the transport level and the lack of competition. In other words, the nodes communicate directly with each other, and the nodes are given the check periodically, in sequential order, so that they can perform arbitrary actions. It should be noted that it is relatively easy to migrate any cycle-based simulation to the event-based engine.





The principal difference between the two models is the different management of time; in cycle-based it is given in "execution cycles" while in event-based in "time values", made out of "time units". Once you have chosen the unit of measurement of time (the one of *simulation.endtime*) you will have to remain consistent with, it every time you need to insert or manipulate a time value.

PeerSim has been designed to encourage modular programming based on objects. Each block is easily replaceable by another component that implements the same interface (that is, the same functionality). The general idea of the simulation model is:

- choose a network size (number of nodes)
- choose one or more protocols to experiment with and initialize them
- 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)
- run your simulation invoking the Simulator class with a configuration file, that contains the above information

The lifecycle of a simulation is as follows. The first step is to read the configuration file, given as a command line parameter or from the Editor. The configuration contains all the simulation parameters related to all the objects involved in the experiment.

Then the simulator sets up the network by initializing the nodes in the network and the protocols contained within them. Each node has the same types of protocols, that is, the instances of a protocol form an array in the network, with an instance in each node. The instances of the nodes and protocols are created by cloning. Only one instance is generated using the constructor object, which acts as a prototype, and all the nodes in the net are cloned by this prototype. For this reason, it is very important to pay attention during the implementation of the protocol cloning method.
If in the nodes and the protocols are present variables of non-primitive type, the cloning should be made with deep clone.

At this point, it is necessary to perform initialization, which sets the initial states of each protocol. The initialization phase is performed by the Controls scheduled for execution only at the beginning of each experiment. In the configuration file the initialization components are easily recognizable by the init prefix. These initial objects are simply Control, configured for execution in the initialization phase.

After initialization, the engine calls all the Protocol and Control components once in each cycle, up to a certain number of cycles until a component decides to end the simulation.
In Peersim, all the Protocol and Control objects are assigned to a Scheduler object that defines when they will be executed. By default, all objects are executed in each cycle. However it is possible to configure a protocol or a control for execution only in specific cycles and it is also possible to choose the order of execution of the components within each cycle.

In the event-based model, everything works in exactly the same way as the cycle-based model, except for time management and for the way in which control is given to the protocols. *Protocol* that are not executable (used only for storing data) can be applied and initialized in exactly the same way. You can also use the *Control* of any package outside the *peersim.cdsim package*. By default, the *Control* are called in each cycle of the cycle-based model. In the event-based model *Control* must be explicitly scheduled, since there are no cycles.

Once chosen the most congenial view of time, it is necessary to determine how many components are needed to manage the system we want to achieve.
For each component, the periodicity and the connections with the other components must be decided.

The periodicity, as mentioned previously, in the cycle-based model has granularity of the cycle, while in the event-based model it is expressed in time-series units.
This second model therefore allows us to obtain greater realism from this point of view.
For this reason the developer is advised to write the times in a way that makes clear the temporal unit of measure chosen, so as to help the other readers of the code.
For example, by specifying for the duration of the simulation 60 * 60 * 1000, the natural interpretation is that the unit of measurement is in milliseconds. A simulation of a duration of 1 hour is thus assumed, even though the same result could've been achieved by writing 1 and keeping all the others in hours. It would be completely analogous, but the unit of measure would certainly be less "visible".
The number of time units indicating the total duration of the simulation are to be specified using the configuration file, in simulation.endtime.
Also the periodicity of each component is specified in the configuration file
for *Protocol* in protocol.NAME.step and for *Control* in control.NAME.step
and indicates how many units of time the component will activate.
Moreover, for each component it is possible to indicate a start time, different from time 0, and an end of execution time, different from simulation.endtime indicated in
protocol(control).NAME.from and in protocol(control).NAME.until
Alternatively, the component can be set for just one execution at the time indicated in
protocol(control).NAME.at
It should also be noted that it is possible to indicate a time value also for the delay of an event (for example a message between one node and another).

As for the connection between the various components, it is necessary to evaluate each component, if it needs to access data or methods of other component instances.
Classic example: a *Protocol* will need to modify its internal variables or activate its methods with a periodicity (protocol.NAME.step) different from its own, it will be necessary to create a *Control* with the periodicity (control.NAME.step) needed and link it to each instance of the *Protocol*. It is important to remember that for each *Control* only one instance is created (which sees the network as a whole and has access to all nodes) while for each *Protocol* an instance is created for each node.

To connect a component is used this convention:

```java
public class    Observer implements Control {
    // ------------------------------------------------------------------------
    // Parameters
    // ------------------------------------------------------------------------

    /**
     * The protocol to look at.
     *
     * @config
     */
    private static final String PAR_PROT = "protocol";

    // ------------------------------------------------------------------------
    // Fields
    // ------------------------------------------------------------------------

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

    // ------------------------------------------------------------------------
    // Constructor
    // ------------------------------------------------------------------------
    /**
     * Standard constructor that reads the configuration parameters. Invoked by
     * the simulation engine.
     *
     * @param prefix
     *            the configuration prefix for this class.
     */
    public BLEObserver(String prefix) {
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
    }
```

In this way, the pid variable (the id of the component you want to connect) is available.
What we see in the last line of code is the way in which the information from the configuration file is transmitted, and through this the name of the component that needs to be connected will be indicated. Likewise from the configuration file it is also possible to pass numerical or text values, just call the appropriate method (Configuration.getString for a string for example).

In the *Control* you can then access the instances of all connected protocols' nodes.

```java
// Control interface method.
public boolean execute() {


    for (int i = 0; i < Network.size(); i++){

        Myprotocol myprotocol = (Myprotocol) Network.get(i).getProtocol(pid);
        ...
    }

    return false;
}
```

While in the *Protocol* is possible to access the instances of the connected components (always using the pid variable) but only of the nodes of which we have vision, so whose id is known.

```java
Linkable linkable = (Linkable) node.getProtocol( FastConfig.getLinkable(pid) );
Transport transport = (Transport) node.getProtocol( FastConfig.getTransport(pid) );

for (int i = 0; i < linkable.degree(); i++) {

    Node peern = linkable.getNeighbor(i);

    Myprotocol myprotocol = (Myprotocol) peern.getProtocol(pid);
```

The pid of a Protocol will always be the same for all the nodes, this is due to the fact that all the instances of a same Protocol are connected by default, as previously mentioned.

```
Linkable linkable = (Linkable) node.getProtocol( FastConfig.getLinkable(pid) );
Transport transport = (Transport) node.getProtocol( FastConfig.getTransport(pid) );

for (int i = 0; i < linkable.degree(); i++) {

    Node peern = linkable.getNeighbor(i);

    transport.send(node, peern, msg, pid);
```

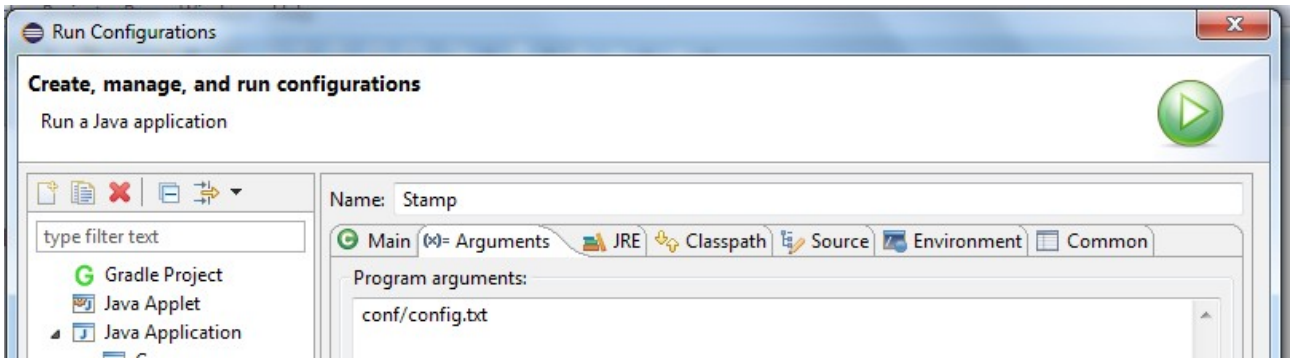
In order to exchange messages between instances of different nodes of a same protocol, you can use the transport layer or directly EDSimulator.add

```
EDSimulator.add(
        delay,
        msg,
        peern,
        pid);
```

while to access the nodes you are viewing, the linkable level can be used, as shown.

Once confidence is gained with the management of these two fundamental characteristics we have all the tools to design your own simulation and then implement your own protocol (on which you want to experiment) and all the classes of the other necessary components.

Finally, remember that even if you want to start the simulation from an Editor, you need to pass the configuration file with the parameter:



During the run of the program, in order to monitor the trend of certain variables (the useful ones to be observed) towards the required periodicity, appropriate *Control*, called Observers, are used. However we also recommend using them to save interest values on file .dat, to keep track of these. In this way you will also have them available for subsequent analysis. They will also be usable without problems on Gnuplot, or they can be easily converted into Excel format.