A Java based framework for simulating peer-to-peer overlay networks

Daniel Hasselrot
daniel.hasselrot@sics.se

SICS Technical Report T2005:02
ISSN 1100-3154
ISRN:SICS-T–2005/02-SE

January 24, 2005

Abstract

In the last few years many new structured overlay network protocols for peer-to-peer systems have appeared. Following that, the need to test and develop the protocols in a controlled environment arose and many different simulators were written, often only supporting a single protocol and designed with a specific simulation task in mind. We introduce a general component based framework in Java for writing peer-to-peer simulators, complete with methods for exporting and visualizing the simulation data. The simulation framework is built using the Java in Simulation Time (JiST) framework, a parallel discrete-event simulation package for Java, allowing close to pseudo code implementation of the protocols using simulated remote procedure calls, while still providing flexibility enough for down to packet level simulation of the protocols.

Keywords: peer-to-peer, simulation, structured overlay networks

1 Introduction

When first introduced, peer-to-peer (p2p) networks only provided ad-hoc methods for locating information on the peers. They either relied on central servers, for example Napster [13], or flooding neighborhood searches, for example Gnutella [6], solutions which do not scale to large networks. To overcome this problem, structured p2p overlay network protocols, such as Chord [18] and Kademlia [12], were introduced. These protocols are capable of keeping correct routing information to all nodes in the network while requiring each individual node only to maintain a small-sized routing table, typically in the logarithmic size of the network. The structure of the overlay networks allows them to scale up to huge networks while maintaining logarithmic lookup times. Generally all structured overlay network
protocols functions as distributed hash tables, relying on the principles of \(k\)-ary-search for locating key-value pairs [3], although their actual implementation and structure may vary significantly. In common of all structured overlay networks is that they, in a dynamic environment where peers join and leave the network arbitrarily, have some form of maintenance procedures for keeping the routing information up to date with the network.

When comparing different overlay network protocols or evaluating changes or enhancement to the protocols, it is either impossible or inconvenient to implement and compare the protocols on real networks. Most published results on p2p overlay networks are therefore based on simulations, often using simulators written from scratch and customized to the authors specific needs, for example [18, 10]. There exist some efforts for writing general structured overlay network simulators supporting different overlay protocols, such as p2psim [15] which supports a variety of protocols. P2psim is implemented in C++ and built around simulated remote procedure calls using lightweight threads. While being fast, C++ is far from the ideal language for writing simulators in, lacking mechanisms for garbage collection, being platform specific and having an inherently complex and error prone syntax. P2psim also, in its current implementation, has no standard way of outputting data and statistics from the simulation, relying on simple `printf` outputs, which complicates debugging and makes it harder to overview the outputted results.

We have implemented a component based framework in Java for simulating structured overlay networks using simulated remote procedure calls. Our framework makes it easy to implement and compare different overlay network protocols and allows for fast prototyping of new structures by providing an architecture that enables sharing of components between protocols and the ability to maintain close resemblance to the protocols pseudo-code implementation while still being flexible enough to support down to packet level simulations. We have also implemented general method for exporting and visualizing data from the simulations, which greatly enhances the understanding of the behavior of the overlay networks and also aids in debugging.

### 1.1 Chord protocol

Although our framework is intended for simulations of any network overlay protocol the examples and figures in this report is based on simulations of the Chord [18] protocol. In the Chord structure, the peers are spread out on a ring, typically based on a unique hash of their IP addresses. Each peer on the ring have pointers to one or a few successors. These successor pointers give enough information for routing in the network but only offer linear worst-case lookup times for the hash keys. To speed up lookups, another structure is used, called finger pointers. These pointers are spread out over the ring with exponentially increasing intervals and are used for fast, logarithmic, routing to distant peers. The maintenance procedure for the chord protocol, when the finger and successor pointers are updated to reflect the changes in a dynamic network, is called stabilization.
2 Simulation framework architecture

The basis of our simulation framework is an discrete-event simulator built using the Java in Simulation Time (JiST) framework [1] described below. The simulator is component based, relying heavily on the object-oriented properties of Java to allow as much reuse of components between different types of protocols as possible. The implementations of the structured overlay network protocols only have to provide a set of minimum functionalities: each peer should be able to look up a hash key, by either returning the answer to the lookup question itself or passing the question on to a more well informed peer, performing a lookup hop, and each peer should provide the ability for a new peer to join the network through them, inserting the new node in the correct place in the overlay network structure.

2.1 Java in Simulation Time - JiST

JiST is an efficient class loader based framework for parallel discrete-event simulations that allows the use of standard Java compilers to write simulators where the event scheduling and handling is transparent to the developer. JiST achieves this by providing some simulation specific semantics which are rewritten into regular Java code during the loading of the classes. One drawback with this approach is that it prevents the use of debuggers as the original source code and the code of the rewritten class may have little in common. However, taking into account that debugging parallel distributed applications using ordinary debuggers in most cases already is impossible, requiring specialized debuggers such as ETNUS [5], and that JiST, when run in debug mode, provides stack traces that clearly identifies the source of any simulation exception, debugging the simulation remains fairly straightforward and impose no major disadvantage compared to other simulation approaches.

A JiST simulation is based on interactions between entities, an entity being a conceptual extension to the regular Java object model. The main difference between an object and an entity is that the entity methods are invoked in simulation time, that is, instead of a method call being executed instantaneously, the call is scheduled on the target entity as an event and executed when the simulation reaches the scheduled time point. Each entity has its own internal time which only can be advanced by explicitly telling the entity to wait (or sleep, using JiST syntax) for a certain number of time steps; all other code takes, in the simulation time sense, zero time to execute. Sleeping does not effect the execution of the code within methods, only at which time a method call is scheduled on its target. Thus, the time and state of the entities are only synchronized upon invocation of one of their methods, allowing for parallel execution, in simulation time, of the entity methods, this without having to use any time and memory consuming multi-threading approaches.

While providing asynchronous method calls that are scheduled in simulation time, JiST also provides synchronous calls, called continuations. Upon invocation
of a continuation method on an entity, the execution of the method containing the invocation is stopped, its call stack is saved\(^1\), and the invocation is scheduled on the target entity as a asynchronous call. The continuation method itself will execute as a regular entity method until it reaches its return statement. Then, it will schedule a return event on its caller, based on the current internal time of the entity when reaching the statement. When that event is executed, the call stack of the halted method will be restored and the execution will continue on the next statement after the continuation method invocation. This allows continuations methods to pass return variables to the caller and admits the overlay network protocols to be written using simulated remote procedure calls (RPC) keeping the code as close to pseudo code as possible, and cleared from any explicit message handling while still providing enough flexibility to do for example down to packet level simulations of the overlay networks.

Other benefits of using JiST are that functions for running simulations in distributed environments using remote method invocation (RMI) is built in and capabilities for scripting simulations are also supported. Simulations can either be scripted in Java (BeanShell [2]) or Python (Jython [9]) and other script languages can easily be added. Using a dynamic language for scripting instead of static parameter passing has one great advantage in that it enables us to use arbitrary classes as components and parameters while still enabling execution of simulations in a distributed environment without having to write any specialized factory classes or parameter parsers, that needs to be updated for each new component.

2.2 Simulation entities

Our simulator is component based and built around four basic entity types: the simulation, the network, the peer and the event generator entities. The simulation entity has responsibility for starting up the simulation and initiating event generators for recording data, writing out status etc. The network entity keeps track of the peers and the network topology. The peer entity is responsible for all lookups of hash keys and all other peer functions relevant to the implemented overlay protocol. The event generators are responsible for generating all the events in the simulation. They can either be attached to the simulation itself or to an individual peer. Typically, generators generating events that do not depend on the number of peers in the network are assigned to the simulation, for example the join event generator, and the peers is assigned event generators initiating peer dependent events such as lookup traffic.

Each hop in a lookup is scheduled through the network entity where the topology component decides how much delay the hop should incur. The delay can either be a concrete delay in simulation time or just an abstract cost for making the hop that is saved for statistics. The topology component could be extended to allow

\(^1\text{Saving the call stack is not possible in standard Java, but JiST works around this by rewriting the methods when loading the class to allow for different starting points each time they are invoked.}\)
for packet level simulation of the traffic generated by the overlay network without having to make changes to the rest of the code.

2.3 Event generators

The event generators are the origin of all events in the simulation. Other events may only be scheduled as a direct result of a generator initiating an event, for example, a lookup for a hash key from the traffic generator will initiate a chain of events among the peers as a result of the RPC calls performed when searching for the peer holding the correct key.

The generators are component based and can initiate any kind of events; there are no predefined types or required functionalities. They can be based on different random distributions, be specific to a certain overlay network protocol or generally applicable to all protocols. The idea is that one picks the generators that suit ones simulation needs from a smorgasbord of provided generators. The current implementation supports the generator types described below.

2.3.1 Static event generators

Static event generators are responsible for scheduling events in intervals according to some provided distribution, independent of the current simulation state. For our simulation we have used the following event generators:

**Poisson distributed joins.** Generate Poisson distributed joining of peers to the network. One join process per simulation is used.

**Poisson distributed traffic.** Generate lookups for random hash keys at exponentially distributed intervals. Each peer has its own traffic generator.

**Exponential distributed failures.** Decides the life time of a peer according to an exponential distribution. Only responsible for initiating one event that kills the peer it is attached to.

**Poisson distributed stabilizations.** Generate Poisson distributed stabilization events for the Chord protocol. A stabilization repairs either the finger or the successor pointers in the Chord ring.

There are also event generators responsible for collecting data from the simulation and printing the simulation status to the output console. In JiST the order of the events executed in a single time step is arbitrary and does not depend on the order they were scheduled in. When collecting data, it is essential that all entities in the simulation are synchronized to the same time step and have processed all of their scheduled events before writing the data. But using regular JiST simulation this is impossible as there is no way make sure that an event will be executed last in each time step it is scheduled. To achieve this synchronization we instead provide the possibility to schedule events at a finer granularity than one time step. More
precisely, one can choose to schedule events before a time step, the initialization step, at the time step, the action step, and after the time step, the monitor step. All regular events are executed at the action step, the output to the console is scheduled at the initialization step, and the collection of data is scheduled at the monitor step. This increased granularity is transparent to the simulation, and from the simulation point of view, all these steps are executed at the same simulation time.

2.3.2 Adaptive event generators

As an extension to the static event generators, our system also supports event generators that can depend and adapt their behavior to the state of the simulation. They actually consist of two event generators, an adaptive event generator and a controller event generator. The controller event generator is responsible for collecting and acting on information about the current state of the simulation, for example, it might change behavior according to the number of correctly assigned finger pointers in a certain chord peer or the total number of peers in the network. The controller event generator does not itself schedule any events on the entities in the simulation but changes the distribution or frequency of event generation in the adaptive event generator it has been assigned control of. This way, one adaptive event generator can have one of many different controllers assigned to it, making it easy to test and compare different behavior patterns by just changing the controller.

When a peer joins the network it is possible let the behavior of the adaptive event generator to be inherited from the peer it joins on. As the original peer have already spent some time gathering information about the system and adapted its behavior to the current state of the system, starting out using this information is clearly better than an uninformed initialization of the adaptive event generator. Both the state of the adaptive generator and its controller is inherited.

2.3.3 Transactions

Some chain of events may take many simulation time steps to complete. When an adaptive event generator is inheriting its state and behavior from a peer or when collecting statistics that are dependent on the completion of an event chain, for example statistics about number of hops per lookup, it is important that the statistics are consistent and independent of any ongoing events.

In Figure 1, a sequence diagram showing the execution of a lookup is shown. During the lookup, four joins are scheduled and executed on the first peer in the lookup event chain. When each join is performed the state of the peer and the event generator is inherited by the joining peer. The current state of the peer and the event generator is inherited but any outstanding events are not. For the first two joins, any statistics saved by the peer before performing the first lookup hop will cause inconsistency with the event generator, the event generator knows that a lookup was scheduled, and is ongoing, but is not aware of it current status, if any statistics for the lookup has been collected or not. The problem is even greater
when handling the third join, as the join is scheduled between the scheduling of return event from the continuation method and its execution on the event generator, resulting in that the event generator believes that there is an ongoing lookup taking place but the peer considers it finished. Only the fourth join will have a consistent view of the lookup between the peer and the event generator. Note that the three last joins are scheduled on the same time step and still have different outcome. This is because of the undefined order of execution of events that are scheduled at the same time step.

To overcome this and achieve consistency under all circumstances, we use the same principles of transactions that are commonly used in databases, where a sequence of changes to the database are only actually stored in the database when and if all changes proceeded successfully. In our simulation, a transaction is a chain of events generating statistics that is only committed to the simulation statistics upon successful completion of all events. This allows for example statistics collected for a lookup that eventually fails to be rolled back and discarded. In the example in Figure 1, the whole chain of lookup hops is considered as one lookup transaction. Statistics from the lookup are not updated until the transaction is finished and the result returned to the event generator. For the first three joins, the status of both the peer and the event generator will be that there is an ongoing event, but no statistics has been saved. Only the last join will inherit statistics from the lookup, as it take place when the transaction is finished.

Figure 1: A sequence diagram of scheduled events during a hash key lookup interrupted by four joins.
2.3.4 Macro time lottery scheduling of events

The event generators and JiST can also be used to implement other time models than the discrete-event time. Some analytical approaches for analyzing the behavior of the network requires other types of time models, for example El-Ansary et al. [4] uses lottery scheduling instead such that at each time step, each peer has a probability of either having a node joining on him, to fail or to perform a stabilization of either his successors or fingers. The difference here lies in that at every time step an action has to be executed; changing the probability of the different actions will change the actual time spend at each time step and the notion of time in the simulation.

Our simulation framework also support the concept of lottery scheduling, albeit on a basic level, by using a simulation with a single event generator, that at each time step activates the lottery scheduler, which then is responsible for initiating the actions on the different peers. Each of the actions are performed instantaneously and are not scheduled by JiST. Even though much of the functionality of JiST remains unused for this time model, we still have the advantage of being able to reuse the same peer and network classes without making any changes in their code, and also maintaining a consistent interface to the simulation, regardless of time model.

3 Collecting hierarchical simulation data

Most simulation kernels and libraries have some functionalities for gathering data from the simulation, to enable generation of graphs and statistics. Peer-to-peer simulators in general, such as the previously mentioned p2psim [15], are often based on custom simulation kernels, written from scratch, and supporting data extraction on the most basic and ad-hoc level. The JiST framework itself only provides a simple log interface for writing out debug information and data from the simulation. One drawback with using the logging approach for writing out data is that it is disconnected from the simulation time and provides no standard way of managing and visualizing the outputted data.

We introduce a flexible and efficient way of gathering data from simulations that is closely coupled with the structure of the objects and entities in the simulation and the simulation time. The idea is to have a structure that allows the attachment of probes to objects which then can be used to gather data from the objects while incurring no overhead to the simulation when no data is gathered. This probe structure can both be used to monitor variables in running simulations or for writing down data for off-line processing.

3.1 Probe structure

There are two main classes of probes: the object probe and the data probe. Object probes do not contain data themselves but may contain data probes and other object
probes; data probes encapsulate the actual data, such as numbers or strings, and can not contain any other objects or data probes. Together they form a tree hierarchy where object probes are the parent nodes in the tree and the data probes are leaf nodes. In Figure 2(a) the top probe structure of a p2p simulation of the Chord protocol is shown. In Figure 2(b), the probe structure of the peer object is shown in more detail. Any object implementing the probeable interface can have a object probe attached to it. The interface only has three requirements: a probable object should be able to return a named probe, it should be able to return a collection of all of its probes, and it should be able to updated its probes with the current value of the objects they are attached to. For the data probes, this can be actual member variables of the object or objects returned by a method.

There are also some specialization classes of the object and data probes. One specialization class is the collection probe that allows probing of general collections of probable objects and is used when for example probing the collection of peers belonging to a network in Figure 2(a). The data probes can also be set to be indexed (marked as squared nodes in Figure 2(a)) which allows for faster lookups when visualizing the data but slows down the export.

3.2 Writing hierarchical data

When writing out data from the simulation the probe tree is treated like a search tree. The tree nodes are processed in a depth first search order and all the data probes are stored in the order they are encountered. The tree is not static but generated dynamically during each write query, allowing for example the number of nodes to change. Several different output formats are available, from XML to database tables, and additional writers for other formats can easily be added.

3.2.1 XML output

The XML format is ideal for writing out hierarchical data and the writer produces a plain copy of the probe tree hierarchy. Although producing nice, readable and exportable data, the overhead of the XML format makes it useful only for outputting very limited amounts of data. In Appendix A an example XML output from a simulation using probes set up like the structure in Figure 2(b) is shown.

3.2.2 Hierarchical tables

A more useful output format is the hierarchical structured tables, which for example can be used to produce plain CSV files for input into a standard spreadsheet application such as Excel, or to produce output to databases. All the probe data is stored in a non-normalized form, that is, instead of storing the different levels of the hierarchy in different tables and joining them on foreign keys, only one table is used and the rows are allowed to contain repeated and redundant information. This is a design choice that allows for fast processing of the data using simple queries.
Figure 2: An example probe structure for a simulation of the Chord protocol.

Any redundancies can be removed after data extraction using provided meta data of the probe hierarchy.

In Table 1 an example hierarchical table is shown, outputted using the same probe data as in the XML example in Appendix A. Although this output is much more compact it has some information that is repeated, for example the time values are repeated for all peers probed at the same place in time. For each variable sized probe collection, where the amount of probes may vary from time to time, all data above the collection in the probe hierarchy will be repeated in the table, one time for each probe in the collection. If, for example, the number of fingers is allowed to differ between the peers, the outputted table would look as in Table 2. Clearly, having a lot of variable sized collections in the probe hierarchy will intro-
duce a massive amount of redundant data in the tables. But by compressing the table, as implemented in for example the hierarchical data format HDF5 [7], this redundancy effect on the size of the table is drastically reduced, as repeated data is compressed with great efficiency. Also compressing the data might improve the speed of loading and saving the data [16]. We have not yet implemented compression or HDF5 output, so the effect on performance in our case is to be determined, but uncompressed hierarchical tables are still useful, at least for processing moderate sized data.

When writing a hierarchical structured table to a database the database table is created upon the first querying of the probes with column names and data types extracted from the data probes. All indexed data probes are converted to database indexes when creating the database table enabling fast random access and operation on the rows containing the indexed variables.

<table>
<thead>
<tr>
<th>Time</th>
<th>Peer ID</th>
<th>Lookups</th>
<th>Alive fingers</th>
<th>Finger1OK</th>
<th>Finger2OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17469</td>
<td>0.0</td>
<td>1.0</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>52451</td>
<td>0.0</td>
<td>1.0</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>17469</td>
<td>236.0</td>
<td>0.5</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 1: An example output using hierarchical tables of the probe structure in Figure 2(a), using static number of finger pointers.

<table>
<thead>
<tr>
<th>Time</th>
<th>Peer ID</th>
<th>Lookups</th>
<th>Alive fingers</th>
<th>FingerOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17469</td>
<td>0.0</td>
<td>1.0</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>17469</td>
<td>0.0</td>
<td>1.0</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>52451</td>
<td>0.0</td>
<td>1.0</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>52451</td>
<td>0.0</td>
<td>1.0</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>17469</td>
<td>236.0</td>
<td>0.5</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>17469</td>
<td>236.0</td>
<td>0.5</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 2: Hierarchical table output using the same data as in Table 1, but allowing variable number of finger pointers.

### 3.3 Querying probes

Gathering data from all probes attached to the objects may lead to a vast amount of data needing to be stored and processed. Different simulations often need to extract different types of data and setting up probes for all simulation needs might generate a lot of excessive data from probes that we, in the current simulation run, have no interest in. Still, we want to avoid the need to make changes to the code when selecting which data to save, having to recompile, and in a distributed environment, redistribute the code for each such change.
Figure 3: Querying out specific data probes in the probe hierarchy.

The hierarchical search tree structure allows for fast querying of interesting probes at run-time without recompilation of the code. This also makes it possible to use the probes for real time monitoring of individual variables during a simulation run, without sending a vast amount of unwanted data or introducing some other type of data structure. In Figure 3, an example query is shown where the lookup of a specific node is queried. The queries allow selection of specific data probes while large parts of the search tree remain unvisited. It also allow us to introduce runtime aggregates, for example the mean number of lookups for the peers, minimizing the amount of data needed to be sent when monitoring a simulation in real time. To speed up querying, several queries can be compiled into a query tree and processed simultaneously.

4 Visualize hierarchical simulation data

The many tools developed for scientific visualization of discrete-event simulation data are primarily concentrated on visualizing 3D or multidimensional data for special purposes such as SimVis for visualizing flow simulation data [17]. Other, standard statistics software mainly provides a static view of the data, as does general tools for numerical analysis such as Octave [14] or Matlab [11], commonly used for graph generation. We have created a simple tool, called MoonViz, for quickly overviewing and visualizing simulation data without having to write any code; providing multiple simultaneous views of the data while allowing play back and random access of the data in simulation time. In Figure 4, a sample screen shot of the tool is shown. Data from multiple simulations runs can also be visualized at the same time enabling comparisons of different protocols and assisting in debugging when developing new protocols or features. The tool is developed to increase
the understanding of the simulated protocols and for debugging, and not intended
for producing graphs of final results for use in papers or reports, although graphs
can be exported using the scalable vector format.

The basis of MoonViz is the hierarchical data format described in Section 3.2.2
and the representation is used internally by MoonViz, but loaders for other for-
mats can be added. The idea is to provide a general tool for visualization without
making any assumption about the type of data provided.

4.1 Graphs

Currently MoonViz supports histograms, showing the distribution of a variable,
see for example Figure 5(b), line diagrams of variables over the domain variable
and something we call snap shot graphs. Snap shots graphs provide an overview
of the data at the current value of the domain variable. Snapshots can be scatter
plots, plotting a variable against any other variable, or any other type of graph.
JFreeChart [8] is used for generating the graph outputs and other graph types can
easily be added.

4.2 Visualization plug-ins

MoonViz provides a model-view-controller interface for writing specialized visu-
alization plug-ins. Any variables or variable collections in the hierarchical tables
can be used to provide data input to the plug-ins data model, and are selected by
mapping variable names to columns in the hierarchical table. In Figure 5(d), a
simple plug-in showing the peers position on the chord ring is displayed, with each
(a) Scatter plot of the number of lookups performed by each peer up to the current selected time point.

(b) A histogram of the lookup distribution at the same time step as in Figure 5(a).

(c) Viewing the ratio of correct fingers over time for a single peer by changing the domain variable to the peer id.

(d) Plug-in showing the chord ring with the peers colored by the ratio of correct finger pointers.

Figure 5: Example outputs from the MoonViz tool: (a-c) graphs, (d) plug-in.

peer colored according to the ratio of correct finger pointers. The variable mapping can be dynamic and controlled by user input and for example in the chord ring plug-in, any data column could be selected for coloring the peers. Its also possible to inherit other plug-ins model, view and/or controller classes, and easily extending their functionalities.
4.3 Changing domain variable

The MoonViz tool is not just able to handle time data. You can let the domain variable be any indexed data probe in the probe hierarchy. Changing the domain from the time variable to the peer ID, for example, allows the use of the same graph and plug-in tools for visualizing data for a single selected peer. In Figure 5(c) this is used for producing a graph of the ratio of correct fingers over time for a selected peer.

5 Discussion and future work

In this report we have introduced a framework for writing p2p simulations in Java using RPC-calls, complete with methods for data extraction and visualization. The presented framework should be seen as a proof of concept rather than a finished application. A lot of functionality, such as realistic topology components for the packet level simulation, development of a probe query language and data aggregation functions for the visualization tool, remains to be implemented. Still, the framework already provides enough functionality for successfully simulating and visualizing the chord protocol, and the framework is designed with extensibility in mind so that other protocols and functions easily can be added.

6 Acknowledgements

The chord protocol implementation used in this report, and the original macro time simulator, was implemented by Sameh El-Ansary. This project was sponsored by Vinnova, the Swedish Agency for Innovation Systems as part of the AMRAM program.
References


A XML output

Sample XML output of the probes illustrated in Figure 2(a):

```xml
<Simulation>
  <TimeStep>
    <time v="0"/>
    <Network>
      <PeerCollection>
        <Peer>
          <ID v="17469"/>
          <lookups v="0.0"/>
          <Fingers>
            <aliveFingers v="1.0"/>
            <finger1OK v="true"/>
            <finger2OK v="true"/>
          </Fingers>
        </Peer>
        <Peer>
          <ID v="52451"/>
          <lookups v="0.0"/>
          <Fingers>
            <aliveFingers v="1.0"/>
          </Fingers>
        </Peer>
        ...
      </PeerCollection>
    </Network>
  </TimeStep>
  ...  
  <TimeStep>
    <time v="100"/>
    <Network>
      <PeerCollection>
        <Peer>
          <ID v="17469"/>
          <lookups v="236.0"/>
          <Fingers>
            <aliveFingers v="0.5"/>
            <finger1OK v="false"/>
            <finger2OK v="true"/>
          </Fingers>
        </Peer>
        ...
      </PeerCollection>
    </Network>
  </TimeStep>
  ...  
</Simulation>
```