DEUS: a Discrete Event Universal Simulator

Michele Amoretti
Distributed Systems Group
Information Technology Department
University of Parma (Italy)
amoretti@ce.unipr.it

Matteo Agosti
Distributed Systems Group
Information Technology Department
University of Parma (Italy)
agosti@ce.unipr.it

Francesco Zanichelli
Distributed Systems Group
Information Technology Department
University of Parma (Italy)
zanichelli@ce.unipr.it

ABSTRACT

Currently available discrete event simulation tools exhibit important limitations, either being too specific, or providing only a partial API and possibly not enough scalability. In this paper we introduce our novel general purpose simulator, called DEUS, which aims at becoming one of the reference tools in the field of complex system simulation. Its essential Java API provides basic interfaces and classes for modelling nodes, events and processes characterizing the structure and dynamics of any complex system. High usability, configurability and memory efficiency are among the strengths of DEUS, as exemplified in this paper by means of the simulator of Chord peer-to-peer systems we implemented with minor coding effort.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]; C.2.4 [**Computer - Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]

1. INTRODUCTION

Complex systems are dynamic systems composed of interconnected parts that as a whole exhibit one or more properties that could not be gathered from the properties of the individual parts. Examples of complex systems are found in nature, such as ant colonies, human economies, climate, nervous systems, cells and living things, including human beings, as well within modern energy and telecommunication infrastructures, ranging from networked embedded systems to large scale peer-to-peer architectures.

Quantitative approaches to the dynamics of complex systems have to consider a broad range of concepts, from analytical tools, statistical methods and computer simulations to distributed problem solving, learning and adaptation. Very often closed-form, analytical evaluation is not feasible, thus simulation remains the only viable evaluation methodology. Basically every simulation model is a specification of a physical system in terms of a set of *states* and *events*. Hence, performing a simulation means mimicking the occurrence of events over time, and recognizing their effects as represented by states. Future events occurrences induced by states must be scheduled (*i.e.* planned). In *continuous* simulation state changes occur continuously in time, while in *discrete* simulation the occurrence of an event is instantaneous and fixed to a selected point in time. Continuous simulation models can be converted into discrete models, which are more easily managed and thus most used. Depending on the characteristics of the system to be simulated, the reliability of the answer required, and many other factors, either *event driven* simulation (in which time jumps from event to event) or *time driven* simulation (in which time proceeds at a constant step) may be more appropriate.

The complex systems in which we are interested are characterized by events that are not guaranteed to occur at regular intervals, and by the lack of a bound on the time step (*i.e.* it should not be so small as to make the simulation run too long, nor so large as to make the number of events unmanageable). For such highly asynchronous systems it is more appropriate to adopt an event-driven simulation [17]. For example, consider distributed computing systems based on the peer-to-peer paradigm, with nodes randomly joining and leaving, but also emergency rescue and crisis management scenarios, where rescuers do not arrive and leave at regular time intervals.

As we discuss in section 2, currently available discrete event simulation (DES) tools have many limitations: some of them are too specific, others have a limited API, others do not scale well. In this paper we introduce our novel general purpose tool, called DEUS, which aims at becoming one of the reference tools in the field of complex system simulation. The essential Java API of DEUS, which can be downloaded from the project site [2], allows developers to implement (by subclassing)

- nodes (*i.e.* the parts which interact in a complex system, leading to emergent behaviors: humans, pets, cells, robots, intelligent agents, etc.)
- events (*e.g.* node births/deaths, interactions among nodes, interactions with the environment, logs, etc.)
- processes (either stochastic or deterministic, constraining the timeliness of events)

After the discussion of related work, in section 3 we describe the structure and the features of the DEUS API, presenting the core package and the first additional package we implemented in order to support the simulation of

SIMUTools 2009 March 2-6, Rome, Italy

a particular kind of complex systems, namely peer-to-peer systems. In section 4 we illustrate a simulator of the well-known Chord overlay scheme, that we implemented using such additional package. We compare its features (coding requirements, performance) with those of the same peer-to-peer system developed with another API (specific for P2P system simulation). Finally, in section 5 we summarize the contribution of this paper and propose future activity lines.

2. RELATED WORKS

An almost complete and frequently updated collection of modelling and simulation resources on the Internet is the one maintained by Rizzoli [9]. It is evident that almost every simulation tool targets a specific class of problems. Only few of them may be considered *general purpose*. Among these, the most advanced, in our opinion, is CD++ [16], which is a modelling environment that allows to define and execute DEVS models [17]. The DEVs formalism describes a real system is described as a composite of submodels, each of them being behavioral (atomic) and structural (coupled). Closure under coupling allows coupled models to be integrated into a model hierarchy. Each model is defined by a time base, inputs, states, output and functions to compute the next states and outputs. In CD++, each new atomic model requires the development of a C++ class derived from *Atomic*, which is provided by the tool itself. With respect to our approach, which defines nodes, events and processes, DEVS and CD++ focus on states and functions that change states of models. We think that the two approaches are complementary. CD++ is a sound tool and has also a GUI for the configuration and execution of the models experiments (otherwise a plain text description should be written). Being DEUS more recent, it still lacks of many facilities that will be added in the near future (see section 5).

The same considerations can be made with reference to OMNeT++ [15], another well-known general purpose discrete event simulation tool, which has been publicly available since 1997. Like CD++, also OMNeT++ is based on the concept of simple and compound modules. The user defines the structure of the model (the modules and their interconnection) using a topology description language called NED. The NED language has an equivalent XML representation, that is, NED files can be converted to XML and back without loss of data, including comments. This lowers the barrier for programmatic manipulation of NED files, for example extracting information, refactoring and transforming NED, generating NED from information stored in other system like SQL databases, and so on. DEUS configuration files, that must be directly written in XML, follow the same principle. OMNeT++ has been used in numerous domains from queuing network simulations to wireless and ad-hoc network simulations, from business process simulation to peer-to-peer network, optical switch and storage area network simulations.

Focusing on modelling and simulation of peer-to-peer systems, we analyzed different simulators and realized that no commonly agreed reference architecture exists yet; additionally, only few P2P-related papers report about the simulation environment used to obtain the presented results. The absence of standards leads to the lack of common analysis instruments and makes impossible to reproduce and verify results with different simulators than those used to obtain the original results.

In order to choose the proper simulation environment to be used as starting point for the development of a P2P-specific package based on DEUS, we evaluated different systems according to a set of criteria similar to those presented in [8]:

- *Simulation Architecture*: the operation and the design of the simulator.
- *Usability*: how easy the simulator is to learn and use.
- *Extensibility*: the possibility to modify the standard behavior of the simulator in order to support specific protocols.
- *Configurability*: how easily the simulator can be configured and with which level of detail.
- *Scalability*: the ability to simulate how a P2P protocol scales with thousands, or more, nodes.
- *Statistics*: how much the results are expressive and easy to manipulate.
- *Reusability*: the possibility to use the simulation code to write the real application.

Our reference tool in the last two years has been *PeerSim* [7], because of its clear design and scalability. PeerSim is completely written in Java and offers a well documented API that enables the simulation of structured and unstructured networks. It supports both a cyclic model and a discrete event mode, even though the latter has many unresolved issues (for example it is not possible to associate randomly scheduled events to randomly chosen nodes). PeerSim enables the implementation of personalized components, such as the so-called observers that export custom statistical indicators on the simulation results. Simulations can be configured by writing a plain text file, defining scheduling and parameter values for each component. Developers can easily access the configuration manager in order to make more customizations. To improve its configurability, we extended PeerSim, as we illustrated in a recent work [1].

In general, the most used system for simulating application level protocols is *ns-2* [13], even if it was originally designed to work at network level. Ns-2 is written in C++ and uses the object-oriented paradigm. It offers a discrete event model and an OTcl [12] interpreter as a front-end. However, as ns-2 models both physical and link substrates with high level of detail, it is not very scalable, that is the maximum network size amounts to few hundred nodes. *Ns-3* is not an extension of ns-2, it is a new simulator. The two simulators are both written in C++ but ns-3 is a new simulator that does not support the ns-2 APIs. Some models from ns-2 have already been ported from ns-2 to ns-3. In ns-3, simulation scripts can be written in C++ or in Python. Ns-3 does not have all of the models that ns-2 currently has, but on the other hand, ns-3 does have new capabilities (such as handling multiple interfaces on nodes correctly, use of IP addressing and more alignment with Internet protocols and designs, more detailed 802.11 models, etc.). Ns-2 models can usually be ported to ns-3 (a porting guide is under development).

P2PSim [6] is a discrete-event simulator for structured overlay networks written in C++. P2PSim supports several peer-to-peer protocols including the recent Koorde and

Kademlia, however the different underlaying network models are implemented with a rather abstract level of detail. The lack of documentation makes it hard to extend P2PSim, whereas its scalability is limited to a maximum of few hundred nodes.

OverlayWeaver [10] is a peer-to-peer overlay construction tool written in Java, that provides a common API for higher-level services and a set of routing algorithms like Chord, Kademlia and Koorde. The toolkit contains a so-called Distributed Environment Emulator which invokes and hosts multiple instances of Java applications on a single computer; due to the threads limits imposed by the Java Virtual Machine, the scalability is limited to few hundred nodes. Unfortunately the emulator does not provide network statistics, thus limiting its utilization as a simulator.

PlanetSim [14] is a discrete-event simulator developed in Java, that offers a layered and modular architecture. Distributed services in the simulator uses the Common API for structured overlays enabling the reusability of simulation code to experimentation code running in the Internet. As for OverlayWeaver, it is not possible to collect statistics from the simulation outputs. PlanetSim offers a network layer wrapper which allows to port the simulation code to real networks like PlanetLab; however, this partial support for network protocols limits the scalability, making PlanetSim able to simulate networks with size in the order of 10^5 nodes.

3. DEUS API STRUCTURE AND FEATURES

Being DEUS a general purpose simulator, we kept basic interfaces and classes separated from more specific ones. By means of subclassing, it is possible to create specific modules for the simulation of any kind of complex system. Moreover, we developed a first extension package related to peer-to-peer resource sharing networks.

The experience we acquired during the development of other simulation in Java (mainly using ns-2 [13] and PeerSim [7]) showed us how difficult is to manage memory when it comes to the simulation of systems with a large number of interacting parts (*nodes*, if systems are described as a graph). Java is an extremely powerful language and the flexibility of its object orientation plus the reflection mechanism make it a prolific field upon which build this kind of project; however the difficulties in managing the garbage collection mechanism requires a good design in the memory management. For these reasons, as we describe in more details later on, DEUS relies on an efficient cloning mechanism: the initial process load configuration objects into memories and new instances of those objects are obtained through deep cloning. This features allows the deletion of objects by invoking their class destructor.

3.1 Simulation objects and behavior

The development of DEUS started from the definition of the basic simulation objects and the design of the configuration procedure, having in mind all the dynamics of complex systems that one may need to simulate. The goal was to achieve high flexibility and usability, allowing developers to specify a section with simulation objects and another one with simulation behavior, maximizing the possibility to reuse components and providing self-validation constraints so that the engine could process the configuration file through reflection and without any further validation. Simulation objects are *events*, *nodes*, *resources*, while simulation behavior is managed through *processes* and *engine* objects.

An *event* represents the base simulation unit, *i.e.* the piece of code that is going to be scheduled by the system. Moreover, as complex systems are made by interacting components, we introduced the concept of *node*, which also corresponds to a data structure collector the event could relies on. Each node can have a set of *resources*, a structured way to represent objects the node can share or use through the event code. The association between events and nodes is given by *process* objects that are responsible for event schedule timing calculation. The *engine* object puts everything together by linking events that are scheduled at the beginning of the simulation.

The simulation behavior follows the standard model of discrete event simulations: initialization of system state variables and clock, scheduling of initial events and, until the ending condition is true, calculation of next clock time and processing of the next event in the scheduling queue. However, few additions have been made to make the model more flexible. For each event is possible to specify whether its execution is *one-shot*, so that the event will be removed from the schedule after its completion, or not, so that the event will be rescheduled according to the timing given by its associated process. Moreover, each event is provided with a listening mechanism over the scheduling process so that the latter will be able to schedule other events, namely *referenced events*, right after the event's execution. The ending condition of the simulator happens once the maximum simulation time has been reached or the scheduling queue is empty.

Unlike other simulation frameworks that allow the time of an event to be specified as an interval, giving the start time and the end time of each event, DEUS allows only the specification of the start time. The engine is currently single-threaded, so it has only one current event, but the parallelism in simulation is given by the maintenance of system state according to the virtual time. In the near future we will provide support for multi-threaded and network-distributed simulation engine.

3.2 DEUS core

DEUS has been divided into packages, each one addressing a specific aspect of the simulation. The root package is

```
it.unipr.ce.dsg.deus
```

and contains the following sub-packages:

- `core` base system components including simulation object interfaces, configuration parser and engine;
- `schema` object model representing the configuration file;
- `util` support classes for simulation engine
- `impl.event` reference implementations of the event object;
- `impl.node` reference implementations of the node object;
- `impl.resource` reference implementations of the resource object;

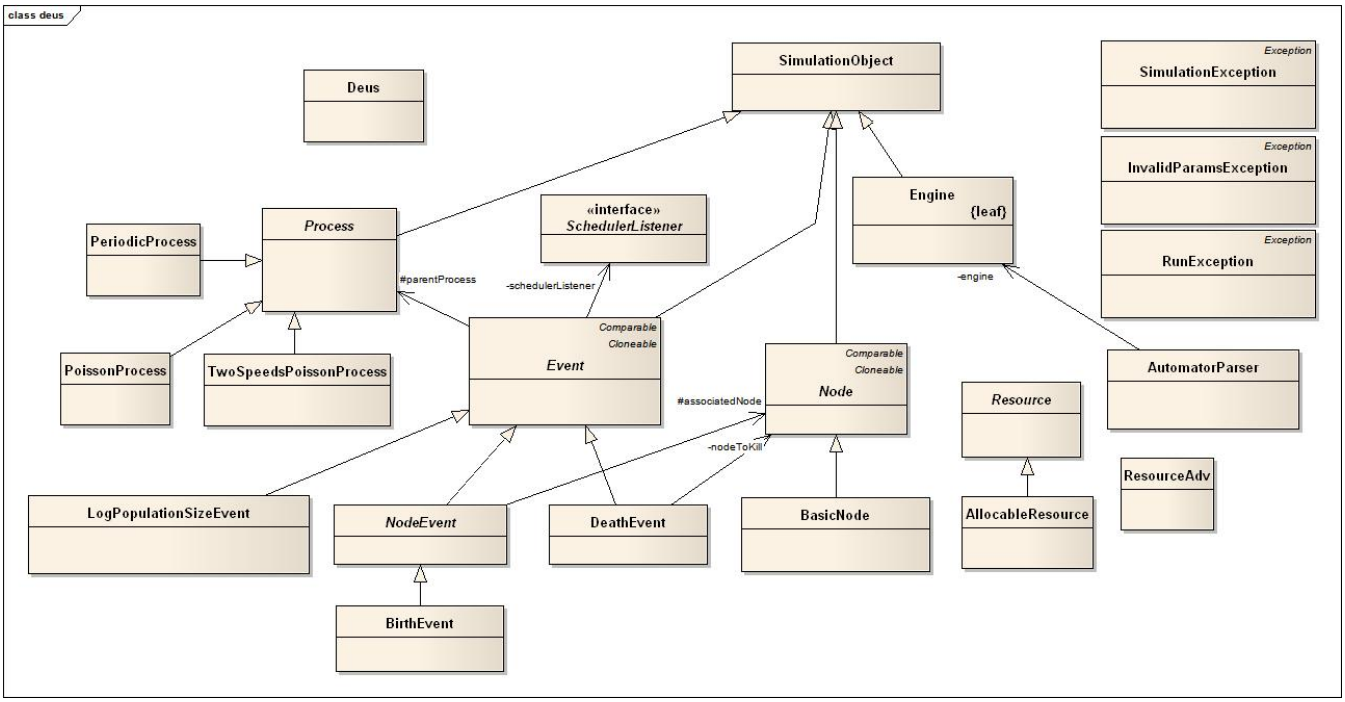

Figure 1: Class diagram of DEUS `core` and `impl` packages.

- `impl.process` reference implementations of the process object.

In the next subsections we will provide a detailed description of the main classes contained in each package, besides the `scheme` and `util` packages that are just used for supporting features. A class diagram including the `core` and `impl` packages is provided in figure 1.

3.2.1 core package

The `Event` class represents the simulation object being scheduled by the Engine. Each event is identified by a configuration id, a set of properties, a flag indicating if the event should be executed only once, a set of referenced events, a parent process, the triggering time and a listener to handle the execution of referenced events. In order to keep the simulation memory area as small as possible, each event is created by cloning the original event obtained from the simulation configuration parser, therefore each implementing class should provide the code for cloning the event ensuring that its internal state is consistent, by re-initializing the event members that do not have to be cloned.

The `Node` class represents a generic data structure collector inside the simulation, so the main use is to store, read and delete information useful for simulation state. Each node is identified by a configuration id, a set of properties and a set of resources. Similarly to the `Event` class there is the same cloning mechanism to keep the memory requirements small for the simulation execution.

The relation between node and event is established with the `NodeEvent` class, used to represents all the events which exist if and only if they are associated with a node; a special flag is used to specify whether the event maintains the same associated node during the cloning process.

The `Resource` class represents a generic resource associated to a node. The class itself does not provide any method, is just used to force implementors to use this node/resource model representation.

The `Process` class represents the simulation object responsible to determine the timing of events scheduling. Each process is identified by a configuration id, a set of properties, a set of referenced nodes and a set of referenced events.

The `Engine` class represents the simulation engine of DEUS. After the configuration file is parsed, the obtained configured simulation objects (nodes, events and processes) are passed to the Engine that will properly initialize the queue of events to be run. The simulation is a standard discrete event simulation where each event has an associated triggering time, used as a sorting criteria. The events inserted into the simulation queue are processed individually one after each other, each time updating the current simulation virtual time. The run method of the engine will process each event in the event queue until a maximum virtual time is reached or the queue is empty. In each cycle the first event of the queue is removed (the one with the lowest triggering time), the virtual time of the simulation is updated and the event is executed. If the event has some referenced events, those will be scheduled right after the event execution, in the same order used to define them in the configuration file. If the event is not one-shot and it has a parent process, then it will be scheduled for a next execution with a triggering time calculated according to the parent process' strategy.

The `AutomatorParser` class is responsible for handling the simulation configuration file according to the *DEUS XML schema*. The configuration can be seen as a set of nodes, resources, events, processes and engine parameters. This class handles the configuration of each simulation object and stores them in a set of array data structures. Each simulation object has a set of base features, plus references to other

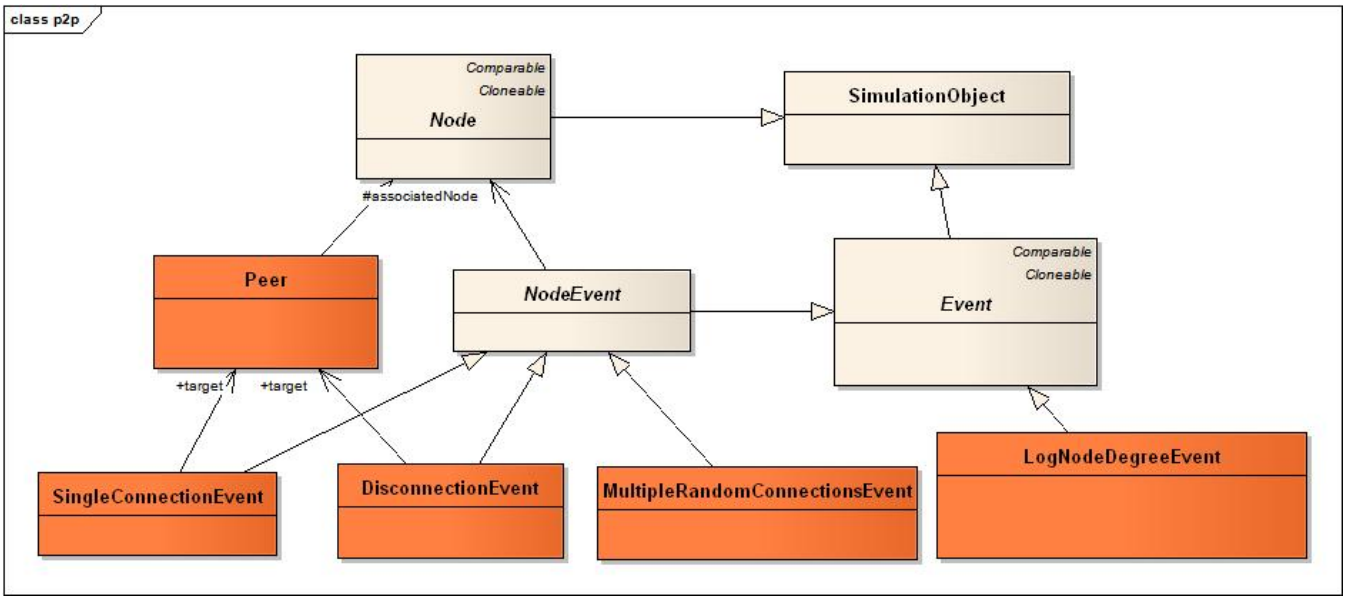

Figure 2: Class diagram of the p2p package.

simulation objects: nodes can have a set of resources, events can have a set of referenced events, processes can have references to both nodes and events. At the end of the configuration file parsing process, this class initializes the Engine object enabling the simulation execution.

3.2.2 *impl.event package*

The `BirthEvent` class represents the birth of a simulated node. During its execution an instance of the node associated to the event will be created.

The `DeathEvent` class represents the death of a simulated node. During the execution of the event the associated node will be killed or, in case nothing is specified, a random node will be chosen instead.

The `LogPopulationSizeEvent` class is used to simulate a logging event that stores the number of nodes in the simulation, each time it is scheduled. It demonstrates that an event can be really anything, in the context of the complex system to be simulated.

3.2.3 *impl.node package*

The `BasicNode` class is the default implementation of the node abstract class, without any specific properties. A specific implementation is provided in the p2p package, which is described later in the paper.

3.2.4 *impl.resource package*

The `AllocableResource` class represents a generic allocable resource. This kind of resource has a type/amount pair parameter which must be specified through the configuration file.

The `ResourceAdv` class represents a resource advertisement, *i.e.* a document that describes a `ConsumableResource` (with a name and an amount), and the interested node. Once the resource described by a `ResourceAdv` has been discovered, the owner of the resource should be registered into the `ResourceAdv`, and the found flag set to true.

3.2.5 *impl.process package*

The `PeriodicProcess` represents a generic periodic process. It has a parameter called *period* that is used to generate the triggering time. Each time the process receives a request for generating a new triggering time, it computes it by adding the period value to the current simulation virtual time. An extension of this class is provided through the `TwoSpeedsPeriodicProcess` class that allows the specification of two different periods; the switch between first period and second period is made using a virtual time threshold.

The `PoissonProcess` represents a generic Poisson process. It has one parameter called *meanArrival*that is used to generate the triggering time. Each time the process receives a request for generating a new triggering time, it computes it by adding the current simulation virtual the value of an Homogeneous Poisson Process with the rate parameter calculated as $1/meanArrival$ time. Similarly to the `TwoSpeedPeriodicProcess`, there is the `TwoSpeedPoissonProcess` class to provide a Poisson Process that changes its speed after a virtual time threshold has been reached.

3.3 Extension package for the simulation of peer-to-peer systems

To simulate a particular kind of complex system, namely peer-to-peer resource sharing networks, we implemented the

```
it.unipr.ce.dsg.deus.p2p
```

package, which contains the following sub-packages:

- `node` the model of peer.
- `event` the events characterizing a P2P network;

In the following we provide a detailed description of the main classes contained in each package. The related class diagram is illustrated in figure 2.

3.3.1 node package

The `Peer` class is an extension of the `Node` class that represents the concept of peer in a network. Each peer is identified by a unique key generated by the engine (in the given key space) and is characterized by a list of neighbors, peers with whom it has an active link connection, and a status regarding peer connection to the network (whether is connected or not). Some methods have been implemented to manage neighborhood and notification messages.

3.3.2 event package

The `SingleConnectionEvent` class simulates the connection event of a peer in the network. The peer can choose a randomly node to which connect or starts from a well-known one. An extension of this class is provided through `MultipleRandomConnectionsEvent`, allowing the connection to more than one node, randomly chosen in the network.

The `DisconnectionEvent` class is used to disconnect a specific node from the network. Alternatively it can be used to disconnect a random node from the network.

The `LogNodeDegreeEvent` class represents a logger that works out on Peer nodes. It calculates the node degree distribution for each peer of the network. The results is a list of degree starting from 1 up to the maximum degree inside the network; for each degree the number of nodes having it is computed.

4. IMPLEMENTATION OF A CHORD SIMULATOR WITH DEUS

The simulative approach is becoming the most common technique to study overlay networks and peer-to-peer applications. The cost of implementing a solution into a simulation environment is considerably lower than what is required to realize a similar experiment on geographical networks. Specifically, the number of computational resources needed is lower and the simulated model can be built to be more realistic than any other tractable mathematical model. The use of a simulation environment may enable the detailed evaluation of architectural models and allow for an high reuse of code when the devised solution will be experimented in the real world.

In the last five years we have been working in the design and development of peer-to-peer middleware [3], and also on the simulation of existing or newly created peer-to-peer protocols [4]. Thus, the decision of creating DEUS was *also* driven by the need for a more flexible simulation tool to support our studies on peer-to-peer architectures. As described in previous section, Java interfaces and classes that are specific for peer-to-peer systems have been grouped in packages that are separated from the core of the tool. In this section we illustrate how we used these building blocks to implement the Chord protocol [11]. By using the reflection to auto build the methods prototype code, the only instructions we had to write are those related to method implementation. In the following, when we say that a class has x code lines, we refer to the x non-automatically generated code lines.

Chord [11] is probably the most known peer-to-peer protocol based on the Structured Model (SM), which uses *Distributed Hash Tables (DHTs)* as infrastructures for building large scale applications. Data are divided into *blocks*, each one identified by a unique *key* (a hash of the block's name) and described by a *value* (typically a pointer to the block's owner). Each peer is assigned a random ID in same space of data block keys, and it is responsible for storing key/value pairs for a limited subset of the entire key space.

Given a key (*i.e.* the identifier of a resource or a service), the Chord protocol maps the key onto a node (a host or a process identified by an IP address and a port number). Chord's consistent hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. Identifiers are ordered on an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal or follows the identifier of k. This node is called the *successor node* of key k. Figure 4 shows a Chord ring with $m = 4$.

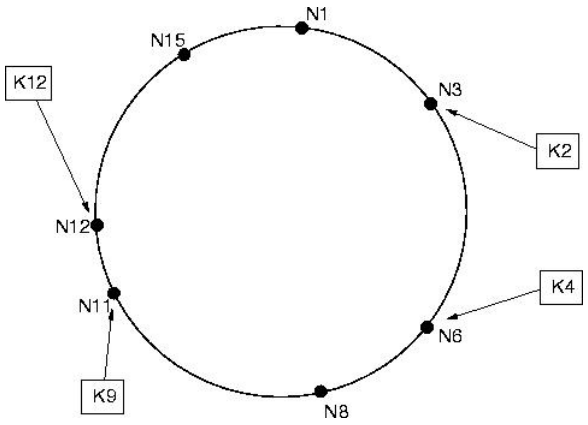

Figure 4: An identifier circle consisting of 7 nodes storing 4 keys. The successor of identifier 2 is the node with identifier 3, so $K2$ is located at $N3$. Similarly for the other 3 keys.

Chord's basic lookup algorithm, whose description we omit for space reasons, uses a number of messages which is linear in the number of nodes. To accelerate lookups, each node n could maintain a routing table with up to m entries, called the *finger table*. The i^{th} entry in the table at node n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle; i.e. $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$ and all the arithmetic is module 2^m. We call node s the i^{th} *finger* of node n, and denote by $n.finger[i]$. A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Figure 5 illustrates the scalable lookup algorithm based on finger tables. In general, if node n searches for a key whose ID falls between n and its successor, node n finds the key in its successor; otherwise, n searches its finger table for the node n' whose ID most immediately precedes the one of the desired key, and then the basic algorithm is executed starting from n'. It is demonstrated that, with high probability, the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$.

The class diagram in figure 3 emphasizes Chord-specific classes with green color.

`ChordPeer` (200 code lines) extends `Peer` (which is defined in the previously illustrated `p2p` package), and pro-

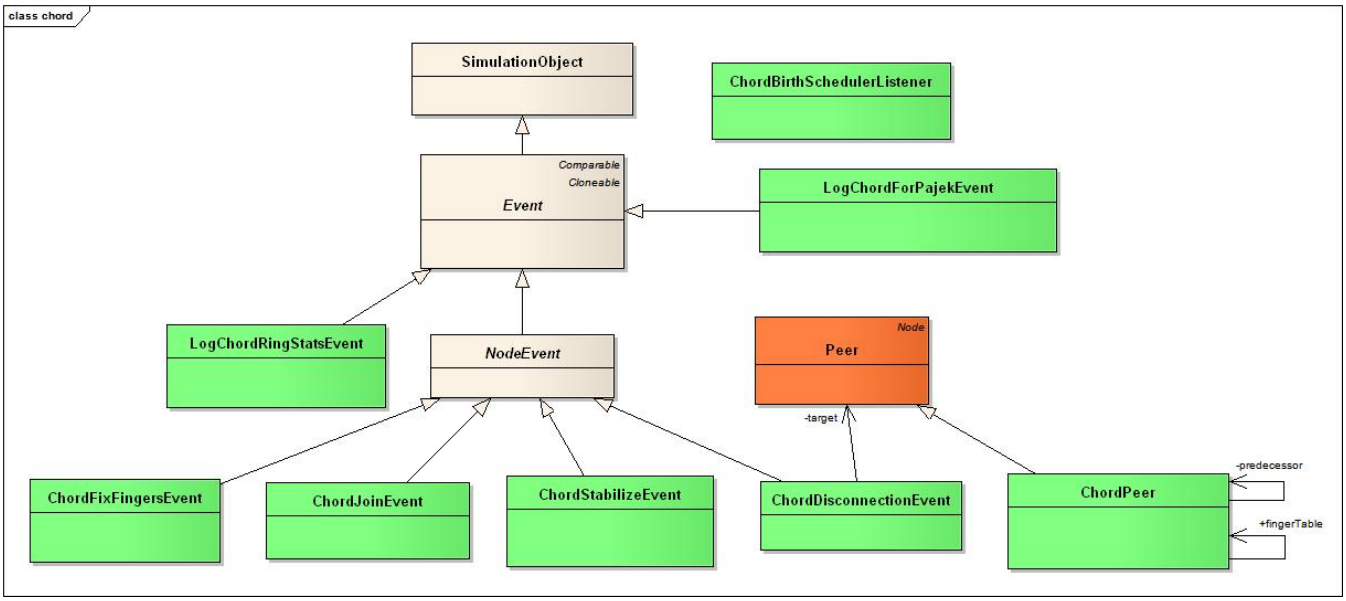

Figure 3: Class diagram of the Chord simulator based on DEUS.

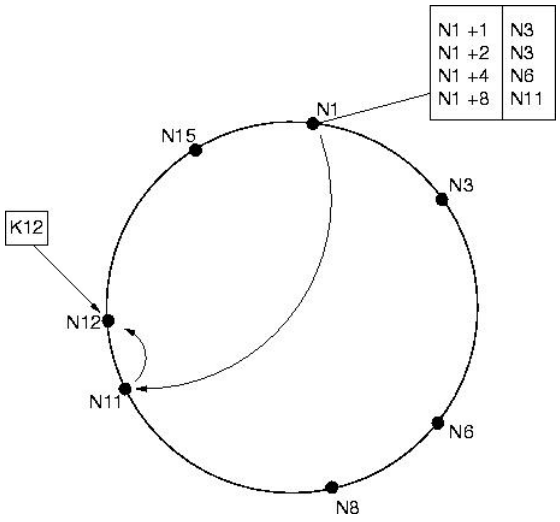

Figure 5: The finger table entries for node $N1$ and the path taken by a query from $N1$, searching for $K9$ using the scalable lookup algorithm.

vides methods for accessing the finger table: `findSuccessor`, `findPredecessor`, etc. These methods map one-to-one with the functionalities defined by the Chord protocol, as it is presented in [11].

`ChordJoinEvent` (17 code lines) is an extension of `NodeEvent` that implements the procedure that each node must execute when joining the network, *i.e.* storing the identifier of the predecessor and initialize th finger table, and updating fingertables and predecessors' identifiers of peers that were in the network when the new node joined.

Each new peer uses a randomly chosen node, among those already connected, to obtain initial information which are necessary to initialize its state and complete the procedure presented above.

There is also `ChordDisconnectionEvent` (12 code lines), that manages the disconnection of a peer, also performing the required adjustments in the finger tables of its predecessor and successor.

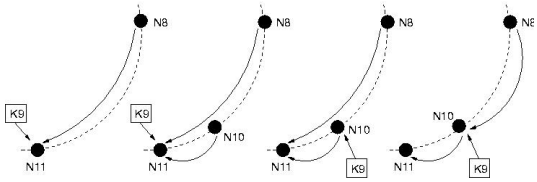

Figure 6: Example illustrating the join operation, including the stabilization procedure.

For each lookup, a `ChordLookupEvent` (20 code lines) created and associated to a destination peer, computed from the finger table of the search initiator. If the requested key is not found in the destination, a new `ChordLookupEvent` is created and associated to the peer which is the new target of the propagation, according to the scalable lookup algorithm described above.

In order to ensure that lookups execute correctly as the set of participating nodes changes, Chord introduces a stabilization protocol that each node should run periodically in the background and which updates finger tables and successor pointers. If any sequence of join operations is executed interleaved with stabilization, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network. In other words, after some time each node is able to reach any other node in the network by following successor pointers. Moreover, if we take a stable network with N nodes (stable means that all successor and finger pointers are correct), and another set of up to N nodes joins the network, then lookups will still take $O(\log N)$ time with high probability.

In our Chord simulator based on DEUS, the stabilization protocol is implemented by the `ChordStabilizeEvent`. It

required to write just 1 code line, because its `run` method simply needs to call the `stabilize` method of `ChordPeer`, whose duty is to ensure that information in each node's successor is always correct, in order to make every network operation successful. Of course the `ChordStabilizeEvent` must be associated to a `PeriodicProcess` when configuring the Chord simulation.

The robustness of the Chord protocol relies on the fact that each node knows its successor. However, this assumption can be compromised if nodes fail. To increase robustness, each Chord node maintains a *successor list* of size r, containing the node's first r successors. If a node's immediate successor does not respond, the node can query the second entry in its successor list. If the length of the successor list is $r = \Omega(\log N)$, in a network that is initially stable, and then every node fails with probability 1/2, then with high probability the procedure to find a successor returns the closest living successor. We implemented an event, namely `ChordFixFingers` (1 code line), which must be scheduled periodically and corrects errors in fingertables of randomly chosen `ChordPeer`s (by invoking their `fixFingers` method).

Finally, we added two logging events:

- `LogChordForPajekEvent` (5 code lines), which produces an output file with node connections, in a format that is suitable for drawing the network topology with Pajek [5].
- `LogChordRingStatsEvent` (8 code lines), that logs on a file, for each connected peer, its predecessor, its successor and its fingertable.

We compare our Chord simulator based on DEUS with the publicly available Chord simulator based on Peersim [7]. In the latter, there is a `ChordInitializer` for the initialization of peers. The core of the simulator is a large class called `ChordProtocol` (300 code lines) which extends `EDProtocol` (allowing the class to process incoming messages), and provides all the functionalities that in our implementation have been splitted into separated events. Since Peersim requires that messages among peers are explicitly modeled, there is a `ChordMessage` interface and different classes that implement it.

PeerSim was designed to encourage modular programming based on objects (building blocks) rather than events. The general idea of PeerSim is:

1. choose a network size (number of nodes);
2. choose one or more protocols to experiment with and initialize them;
3. choose one or more `Control` objects to monitor the properties the user is interested in and to modify some parameters during the simulation (*e.g.*, the size of the network, the internal state of the protocols, etc);
4. run simulations invoking the `Simulator` class with a configuration file, that contains the above information.

Moreover, PeerSim supports two simulation models: cycle-based and event-based.

The simplifying assumptions of the cycle-based model are the lack of transport layer simulation and the lack of concurrency. In other words, nodes communicate with each other directly, and the nodes are given the control periodically, in some sequential order, when they can perform arbitrary actions, such as call methods of other objects and perform some computations. The event-based model is limited by the fact that `Control` objects can be scheduled only periodically or at precise virtual time values, and all nodes are always involved. Thus, for example, PeerSim hinders the simulation of network dynamics such as adding and removing peers, according to some stochastic processes.

On the other side, DEUS allows all this because it is focused on events, as clearly illustrated by the following configuration file, which refers to a Chord simulation in which nodes are added according to a Poisson process with average value of $10t$ (at the end the network size is N 10000). Peers are added one by one, and each `birth` event triggers a `join`, `stabilize` and `fixFingers` (whose execution times are set according to the related Poisson or periodic processes). The whole simulation took 17 minutes on a desktop machine with Intel Core 2 Duo @ 3GHz and 4GB of RAM.

```
<!-- *** EVENTS *** -->
<aut:event id="birth"
  handler="it.unipr.ce.dsg.deus.impl.event.BirthEvent"
  schedulerListener="ChordBirthSchedulerListener">
  <aut:events>
  <!-- events to be scheduled on created node -->
    <aut:reference id="join"/>
    <aut:reference id="stabilize"/>
    <aut:reference id="fixFingers"/>
  </aut:events>
</aut:event>
<aut:event id="join" handler="ChordJoinEvent"
oneShot="true">
  <aut:logger level="OFF"/>
</aut:event>
<aut:event id="stabilize"
handler="ChordStabilizeEvent">
  <aut:params>
    <aut:param name="hasSameAssociatedNode" value="true"/>
  </aut:params>
</aut:event>
<aut:event id="fixFingers"
handler="ChordFixFingersEvent">
  <aut:params>
    <aut:param name="hasSameAssociatedNode" value="true"/>
  </aut:params>
</aut:event>
<aut:event id="logPopulationSize"
handler="LogPopulationSizeEvent"/>
<aut:event id="logChordRingStats"
handler="LogChordRingStatsEvent"/>
<aut:event id="logChordForPajek"
handler="LogChordForPajekEvent"/>

<!-- *** NODE SPECIES *** -->
<aut:node id="chordPeer" handler="ChordPeer">
  <aut:params>
    <aut:param name="fingerTableSize" value="5"/>
  </aut:params>
</aut:node>

<!-- *** PROCESSES *** -->
<aut:process id="poisson1" handler="PoissonProcess">
  <aut:params>
    <aut:param name="meanArrival" value="10"/>
  </aut:params>
  <aut:nodes>
    <aut:reference id="chordPeer"/>
  </aut:nodes>
  <aut:events>
    <aut:reference id="birth"/>
  </aut:events>
```

```
</aut:process>
<aut:process id="poisson2" handler="PoissonProcess">
  <aut:params>
    <aut:param name="meanArrival" value="0"/>
  </aut:params>
  <aut:nodes>
    <aut:reference id="chordPeer"/>
  </aut:nodes>
  <aut:events>
    <aut:reference id="join"/>
  </aut:events>
</aut:process>
<aut:process id="periodic1" handler="PeriodicProcess">
  <aut:params>
    <aut:param name="period" value="1000"/>
  </aut:params>
  <aut:events>
    <aut:reference id="stabilize"/>
    <aut:reference id="fixFingers"/>
  </aut:events>
</aut:process>
<aut:process id="periodic2" handler="PeriodicProcess">
  <aut:params>
    <aut:param name="period" value="10000"/>
  </aut:params>
  <aut:events>
    <aut:reference id="logPopulationSize"/>
    <aut:reference id="logChordRingStats"/>
    <aut:reference id="logChordForPajek"/>
  </aut:events>
</aut:process>

<!-- *** SIMULATION *** -->
<aut:engine maxvt="100000" seed="123456789"
keyspacesize="100000">
  <aut:logger level="ALL"/>
  <aut:processes>
    <aut:reference id="poisson1"/>
    <aut:reference id="periodic2"/>
  </aut:processes>
</aut:engine>
```

5. CONCLUSIONS AND FUTURE WORK

In this paper we illustrated a novel discrete event simulation tool, DEUS, which includes a powerful engine, a XML configuration parser and an extensible Java API for the development of complex system simulators. We described the structure and the main features of the basic packages, and an additional package supporting the simulation of peer-to-peer systems.

We presented the capabilities of our tool by showing a Chord simulator we implemented using the P2P-oriented extension package for DEUS, and we compared its features (coding requirements, support for simulation of dynamic scenarios, performance) with those of the same peer-to-peer system simulator developed with the PeerSim API.

Due to its effectiveness and ease of use, DEUS has become essential for our studies on complex systems, *e.g.* peer-to-peer resource sharing and multimedia streaming networks. In the near future we plan to complete the implementation of a graphical tool for configuring simulations (including a a package to support straightforward sensitivity analysis), to investigate a distributed version of the event execution engine, and to develop new extension packages to support the simulation of other kinds of complex systems. In particular, we plan to develop packages for the simulation of robotic swarms, of emergency rescue scenarios, and of scheduling algorithms for real-time embedded systems.

6. REFERENCES

[1] M. Agosti, M. Amoretti, F. Zanichelli, and G. Conte. P2PAM: a Framework for Peer-to-Peer Architectural Modeling based on PeerSim. In *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008), Marseille, France*, March 2008.

[2] M. Amoretti and M. Agosti. DEUS web page. http://code.google.com/p/deus/, 2008.

[3] M. Amoretti, F. Zanichelli, and G. Conte. SP2A: a Service-oriented Framework for P2P-based Grids. In *3rd International Workshop on Middleware for Grid Computing, Co-located with Middleware 2005.*, November 2005.

[4] M. Amoretti, F. Zanichelli, and G. Conte. Performance Evaluation of Advanced Routing Algorithms for Unstructured Peer-to-Peer Networks. In *First International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2008), Pisa, Italy*, October 2006.

[5] V. Batagelj and A. Mrvar. Pajek web page. http://vlado.fmf.uni-lj.si/pub/networks/pajek/, 2007.

[6] T. M. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/.

[7] M. Jelasity, A. Montresor, G. Jesi, and S. Voulgaris. PeerSim: A Peer-to-Peer Simulator. http://peersim.sourceforge.net, 2004.

[8] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *Computer Communication Review*, 37(2):95–98, 2007.

[9] A. Rizzoli. A Collection of Modelling and Simulation Resources on the Internet. http://www.idsia.ch/ andrea/sim/simtools.html, 2008.

[10] K. Shudo, Y. Tanaka, and S. Sekiguchi. An overlay construction toolkit. http://overlayweaver.sourceforge.net.

[11] I. Stoica, R. Morris, D. Liben Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1), 2003.

[12] The Berkeley Multimedia Research Center. OTcl - object tcl extensions. http://bmrc.berkeley.edu/research/cmt/cmtdoc/otcl/.

[13] University of California Information Sciences Institute. NS-2 network simulator. http://nsnam.isi.edu/nsnam/index.php.

[14] University Rovira i Virgili. Planetsim project. http://planet.urv.es/trac/planetsim.

[15] A. Varga and R. Hornig. An Overview of the OMNeT++ Simulation Environment. In *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008), Marseille, France*, March 2008.

[16] G. Wainer. CD++: a toolkit to develop DEVS models. *Software - Practice and Experience*, 32(13):1–46, november 2002.

[17] B. Zeigler, T. Kim, and Praehofer H. *Theory of Modeling and Simulation.* Academic Press, 2000.