Reproducible Testing of Distributed Software with Middleware Virtualization and Simulation

Cyril Briquet
C.Briquet@ulg.ac.be

Pierre-Arnoul de Marneffe
PA.deMarneffe@ulg.ac.be

Department of Electrical Engineering and Computer Science
University of Liège
Montefiore Institute, B37, B-4000 Liège, Belgium

ABSTRACT

P2P Grids are Grids organized into P2P networks where participant exchange computing time so as to complete computational tasks. Evaluating the performance of scheduling algorithms enables one to deploy those that are efficient. Performance is often evaluated experimentally or through simulation because these algorithms (typically heuristics) are too complex to model analytically. Testing the implementation of P2P Grid middleware before it is deployed is also important: Reproducing configurations or conditions that lead to unexpected outcomes is thus valuable.

A P2P Grid environment exhibits multiple sources of failure and is typically dynamic and uncontrollable. Reproducing even basic behavior of Grid nodes in a controllable and repeatable manner is thus exceedingly difficult. Such lack of control over the environment is a major challenge in the software engineering of P2P Grid middleware [7]. Simulators have been proposed to evaluate the performance of scheduling algorithms, but are often limited in scope, reusability and accuracy, i.e. they rely on simplified models.

We introduce a software engineering pattern - that we call *code once, deploy twice* - to both reduce the distance between simulated and implemented algorithms and reproduce, at will, Grid configurations and environments: A simulator implementation of a Grid architecture is built by virtualizing its middleware implementation. An immediate benefit is that most of the code can be reused between both implementations; only communications between Grid nodes, multithreading within Grid nodes and actual task execution are coded differently. As a derived benefit, most of the code of the middleware can be tested within the controlled environment of the simulator, before it is deployed as-is. Another benefit is high simulation accuracy. We describe the implementation of a P2P Grid following the *code once, deploy twice* pattern, that we believe is also relevant to other Grid types (certainly Volunteer Grids [5, 4] and Desktop Grids [22], and possibly Globus-based Grids [3]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PADTAD'08, July 20–21, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-60558-052-4/08/07 ...$5.00.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.13 [**Software Engineering**]: Reusable Software; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Design, Measurement, Performance, Reliability

Keywords

code reuse, distributed testing, P2P Grid, performance evaluation, scheduling, simulation, virtualization

1. INTRODUCTION

Grid scheduling algorithms are typically evaluated experimentally according to the values of metrics such as mean BoT (Bag of Tasks) response time, system utilization, preemption history, ... Feeding synthetic or trace workloads to scheduling algorithms running in a controlled environment, enables the observation of the behavior and interactions of Grid nodes in an efficient, controllable and reproducible way.

Simulation allows one to run a whole Grid in a controlled environment, running on a single computer rather than on many real computers communicating over the Internet. The purpose of a discrete-event system simulator is to provide a controlled environment where the environment of Grid nodes as well as their time-consuming operations - such as task execution - are abstracted. A simulator takes a simulation description file as input, lets Grid nodes interact and provides execution statistics as output. The temporal cost of simulation can be very small compared to the real execution of a system. The time to simulate many hours of execution time can be compacted down to a few minutes.

We propose a software engineering pattern to develop Grid middlewares and simulators that we call *code once, deploy twice*. This pattern consists of building the simulator implementation of a Grid architecture by virtualizing its middleware implementation. It is based on the virtualization of the Grid middleware itself at the middleware level (not at the hardware or operating system level), which corresponds to the virtualization of the Fabric, Connectivity and Resource layers [17] in Foster et al.'s Grid architecture.

A discrete-event simulator is embedded directly into the Grid middleware. This contrasts with the typical approach that consists of including components (as-is or more often simplified) to simulate into the simulator. Both the middleware and the simulator use essentially the same code, including the scheduling code. Only some operations (interactions between Grid nodes, multithreading within Grid nodes, and actual task execution) are abstracted in the simulator.

This massive code reuse results in high simulation accuracy as well as in several software engineering benefits: Unnecessary software engineering efforts are avoided; The middleware and the simulator can be shipped together and deployed from the same software package. An algorithm that is available in the simulator is also immediately available in the middleware. Testing Grid middleware is thus greatly facilitated.

The *code once, deploy twice* pattern has been used to implement a recent P2P Grid architecture, LBG [9]. We believe that the pattern and implementation are relevant to other Grid types (Globus-based, Volunteer or Desktop Grids) processing long-running computational tasks. Moreover, our approach is complementary with - and does not replace - other forms of testing such as enforcement of invariants, unit testing, automatic verification, static analysis [1, 18] or run-time monitoring [1] and testing of multi-threaded code [13].

The rest of this paper is structured as follows. Section 2 provides a solid background on P2P Grids and discrete-event simulation. Our implementation in Java J2SE 5.0 of a P2P Grid architecture [9] following the *code once, deploy twice* pattern is covered in Sections 3 and 4. Section 3 explains how to virtualize Grid nodes following the *code once, deploy twice* pattern. Section 4 describes our proposed discrete-event P2P Grid simulator. Section 5 introduces a distributed testing process and shows how self-bootstrapping can be achieved. Section 6 presents experimental results. Section 7 discusses related work. Finally, Section 8 summarizes our contributions and discusses future work.

2. BACKGROUND

2.1 P2P Grids

2.1.1 Bartering

A P2P Grid is composed of peers, resources and user agents (see Figure 1). Resources are worker computers that run tasks, and thus generate computing time. Because of the nature of P2P Grids, involved resources are running on so-called edge computers. They thus typically exhibit degraded performance and intermittent availability due to both task execution failure and preemption.

A peer manages a set of resources on behalf of user agents. Peers first use their own resources to compute tasks submitted by user agents. Peers can also barter computing time with one another at the task level. In a P2P Grid context, bartering [12, 9] is defined as distributed, non-monetary exchange of computing time. At peak time, a peer can thus consume computing time from other peers, and supply it back later, at times of low demand levels. A P2P Grid is fully decentralized. Each peer autonomously makes its own bartering and scheduling decisions, based on past interactions with other peers (which it must store).

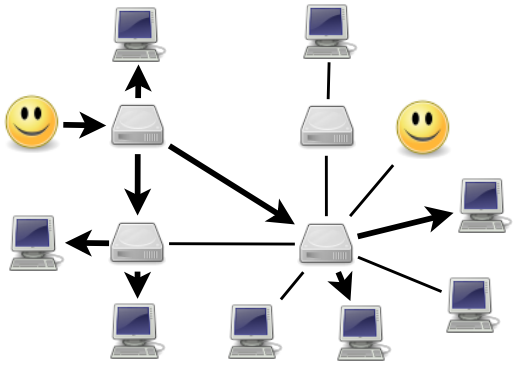


Figure 1: P2P Grid example: 2 user agents, 4 peers, 8 resources.

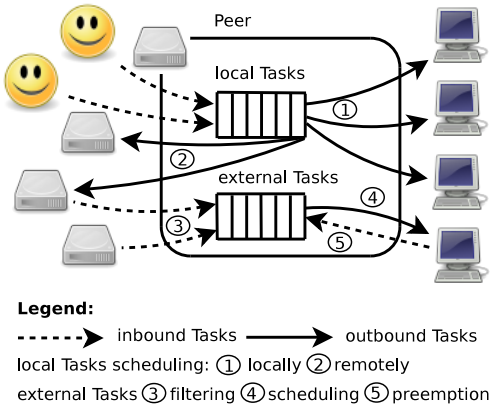


Figure 2: Peer scheduling model and policy decision points. Each peer independently follows this model.

LBG[1] [9] (Lightweight Bartering Grid) is a recent P2P Grid architecture similar to OurGrid [12]. Running the LBG peer middleware or the LBG resource middleware enables to make a computer part of the Grid as, respectively, a peer or a resource. The task model in LBG is the Bag of Tasks (BoT), i.e. an application constituted by a set of independent computational tasks. There is an implicit support for co-allocation in LBG. Peers try to compute their local tasks as fast as possible, requesting access to resources of other peers as needed until their local tasks are completed.

2.1.2 Scheduling Model

Figure 2 illustrates the peer scheduling model and policy decision points. The typical execution cycle in LBG is as follows. User agents submit BoTs to peers. Each peer schedules these local tasks first to its own resources. As long as a peer has queued local tasks and no available resource, it can ask other peers to supply computing time, i.e. run these tasks as external tasks on their resources.

When a peer accepts to supply computing time to compute an external task, it queues it in a second queue separate from local tasks. Each peer waits for some of its resources to be available before computing queued external tasks. Each peer may preempt the execution of external tasks (i.e. submitted by other peers) whenever it has local

[1] `http://www.montefiore.ulg.ac.be/~briquet/`

tasks to compute. Each peer can filter requests for computations of external tasks so that its external tasks queue does not grow out of control. Each peer is equipped with a task control mechanism that preempts tasks that take too long to complete. Each peer makes its own scheduling decisions independently, so that the P2P Grid architecture is fully decentralized.

Task execution is dedicated at the resource level. At any time, at most one task can be run by a given resource. A highly scalable data transfer architecture [8] ensures the efficient transfer of input data files across the P2P Grid.

2.1.3 Grid Nodes Messaging Model

The Grid Node Messaging Protocol (GNMP) is a simple messaging protocol for message passing between Grid nodes. It is based on serialized Java objects transmitted over non-persistent TCP connections. It follows the handle/service pattern (see Figure 3).

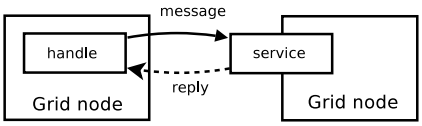


Figure 3: Handle/Service pattern.

Each Grid node is equipped with a service component that processes incoming GNMP messages. Grid nodes send messages through handles, which are Java objects encapsulating the necessary logic and data to communicate with the corresponding service. The handle/service pattern thus abstracts the underlying communication protocol.

2.1.4 Application Programming Model

The LBG middleware supports Java-programmed tasks. In practice, any J2SE 5.0 Java application can be easily prepared to be run on the Grid. One Java class is selected as an entry point by implementing a given interface. All Java classes must be packaged into a jar file. A Bag of Tasks (BoT) is composed of a properties file designating the jar file of each task, along with input parameters and data files.

Resources run tasks in a dedicated Java Virtual Machine (VM), separate from the middleware. A security policy enforces the sandboxing of task execution, i.e. restricts interactions of the VM with its environment.

2.2 Virtualization Levels

Virtualization is the injection of an abstraction layer between an application and some resources used by that application. It provides *a logical rather than physical view of data, computing power, storage capacity, and other resources ... involving the simulation of combined, fragmented, or simplified resources* [27]. The following classification of virtualization levels is inspired by Casanova et al.'s [11]; real executions on real platforms, without modification, are not considered because they are uncontrollable by their very nature.

2.2.1 Virtual Machines and Emulators

At a low level, the hardware environment of the system is completely or partially abstracted. Virtualization technologies operate at this level. Resources run unmodified in a virtual machine (VM) that is controlled by a virtual machine monitor (VMM), or hypervisor.

This is actually emulation and does not require any modification of the studied system. The drawback is that it can be very slow, as the system operates at nominal run-time speed. Simulating one hour of operation of a system takes about one hour, which is unacceptable for Grid operations that span many hours.

2.2.2 Discrete-Event System Simulators

At a higher level, some operations of the system itself are simulated. The system is run, and controlled, by a discrete event system simulator. System simulation within a simulator can be fast, as most time-consuming operations can be abstracted. Communication between system components is very fast because they run together, using the same memory heap. Simulating one hour of operation of a system may be as fast as a few minutes or even seconds.

2.2.3 Mathematical Simulation

At the highest level, most operations of the system are abstracted with an analytical model. The system is controlled by a simple simulator. On one hand, system dynamicity is difficult to take into account, and complex systems are complex to model. On the other hand, it is very fast... when a model is available, which is typically not the case for Grids.

2.3 Discrete-Event System Simulation

2.3.1 Basic Simulation Concepts

Discrete-event system simulation [6] is the modelling of a system over time through its state and a sequence of events. A system is a set of entities interacting with one another, e.g. Grid nodes. The system state is a set of variables, e.g. internal state of Grid nodes. A simulator event represents an asynchronous change in system state and is associated with a timestamp. Statistics on the state of the simulated system are collected and constitute the output of the simulator.

There exists three classical formalisms [19]. Activity Scanning *"is a form of rule based programming, in which a rule is specified upon the satisfaction of which a predefined set of operations is executed."* [19] With a Process Interaction formalism, *"each process in a simulation model specification describes its own action sequence."* [19] With the Event Scheduling formalism, events are defined *"at which discontinuous state transitions occur"* and *"can cause, via scheduling, other events to occur."* [19]

2.3.2 Event Scheduling

Event Scheduling is the most appropriate for the software engineering of P2P Grids. The simulation problem can be expressed in a natural way and simulators based on this formalism are usually faster, although harder to implement.

The event list is the data structure that maintains future events ordered by increasing timestamp. Operations of a discrete-event system simulator based on Event Scheduling (in the following: simulator, for short) are organized around the management of an event list, typically with the Event Scheduling/Time Advance algorithm [6]:

1. The main loop of the simulator sequentially extracts events at the head of the event list;

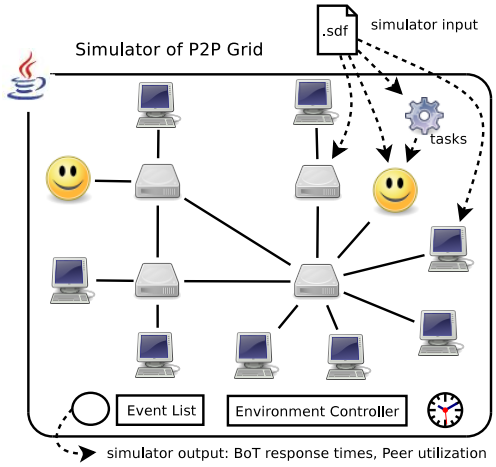


Figure 4: Discrete-event simulator of P2P Grid.

2. When an event is extracted from the event list, the simulator updates the global system time to the value of the timestamp of this event;

3. To process an extracted event, the event processor updates the system state and may insert new events into the event list, in the correct temporal order.

A simulation is started by initializing the system state and time, and also by inserting one or more initial events into the event list. The simulator then enters into its main loop to process events one by one. A simulation could run forever as long as it is fed new events. Criteria to stop simulation include [27]: the system time has exceeded a certain value; the number of events inserted into the system has exceeded a certain threshold; some (possibly indirect or composed) measure of system state has reached a certain value.

Figure 4 illustrates a simulator of P2P Grid. Its input is a file describing the computational power and topology of the Grid to be simulated, the configuration of the Grid nodes and the synthetic workloads to be submitted. Its output is a set of simulation statistics.

3. GRID NODES VIRTUALIZATION

Simulated Grid nodes are instantiated during the initialization of our proposed Grid simulator (see Figure 4). The code of these simulated Grid nodes is loaded (using the Java VM class loader) and shared among all instances, but each simulated Grid node has its own separate data structures (those of peers grow over time due to the storage of metadata about interactions with other peers, as explained in Section 2.1.1).

The code of the simulator implementation of Grid nodes is identical to the code of the middleware implementation. However, some parts of the code have two distinct implementations: One is activated in the simulator, the other one is activated in live Grid nodes. The dual-implemented Java classes are those involved in communications between Grid nodes, multithreading activities and task execution. The simulated Grid nodes have to be virtualized, i.e. isolated from their environment, so that they are not aware of running within the same thread of the same Java VM and of interacting with a fully controlled, virtualized environment.

The virtualization of the Grid nodes code from the middleware implementation is described in this section. The virtualization of communications between Grid nodes is first discussed, followed by the virtualization of Grid nodes themselves. The virtualization of multithreading activities is essentially done for peers, while the virtualization of actual task execution is essentially done for resources. Figure 5 illustrates the differences in execution paths in the case of a live Grid and of a simulated Grid.

3.1 Grid Nodes Messaging Virtualization

In the LBG architecture, GNMP messages pass through handles and services (see Section 2.1.3). In the middleware implementation, a handle performs a network call to send a message to the corresponding service. In the simulator implementation, the network call is replaced by a method call as all handles and services reside in the same Java VM; the transmission of simulated GNMP messages is considered to be infinitely fast (which is reasonable given the context of distributed systems with long-running tasks) and does not cause the system time to be updated.

The processing of GNMP messages by peers is identical in the simulator and in the middleware implementations of peers, but varies in the implementations of resources and user agents: Task execution and submission are virtualized in the simulator implementation.

3.2 Peer Virtualization

The middleware and simulator implementations differ only in how multithreading is implemented. In the middleware implementation, many threads (including the service, timer manager, scheduler, negotiator, data management threads) are started when the peer comes online. In the simulator implementation, a time-delayed communication channel for activation signals - the environment controller - is instantiated and no thread is started.

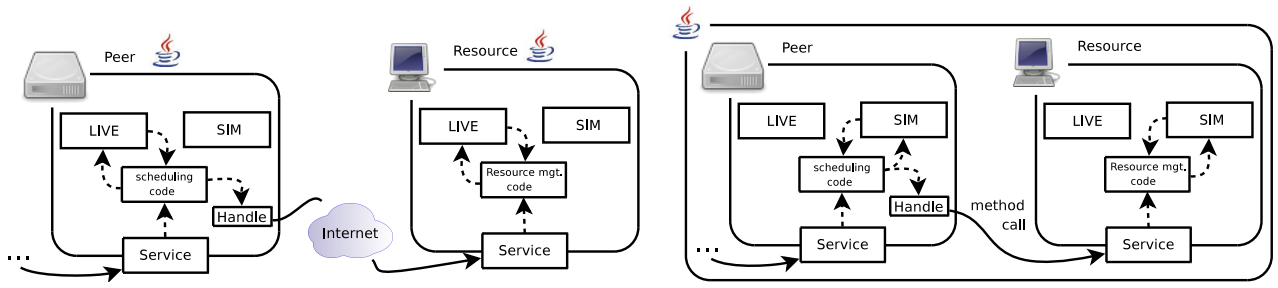
3.2.1 Multithreading Virtualization

In the middleware implementation, many threads are running in every Grid node. Although the service manager (see Section 2.1.3) of every Grid node is simulated, service threads are not. Indeed, a service manager is a purely reactive device. The simulation of helper threads used for task execution and data transfers is straightforward as these operations are abstracted into a very simple model.

The challenge therefore consists of simulating the threads of the scheduler and timer manager of every simulated peer. Running all the scheduling and time management threads within the simulator would be possible but would degrade the simulator scalability. Indeed, the number of threads linearly depends on the number of simulated peers. The schedulers and timer managers should therefore not run within their own, dedicated threads. We propose that the simulator regularly activates every scheduler and timer manager to simulate the multithreading activities of the middleware implementation.

3.2.2 Environment Controller

To simulate the multithreading of the scheduler and timer manager threads, a device called environment controller is introduced. It is a time-delayed communication channel that stores - but does not immediately communicate to the schedulers and timer managers - the activation signals emitted in



(a) Live Grid nodes (middleware implementation)

(b) Virtualized Grid nodes (simulator implementation)

Figure 5: Peer-to-resource interaction illustrating the difference in execution paths between Grid nodes.

other parts of the peer code. The environment controller intercepts activation signals. It precludes the need for an inter-thread communication mechanism, as everything runs in one single thread (the simulator thread).

When a scheduling activation signal is emitted, it is sent to the appropriate scheduler controller. In the simulator implementation, this signal is not immediately forwarded to the scheduler; the state of the environment controller is updated instead. The actual forwarding of an activation signal to the relevant scheduler or timer manager is done when the simulator activates the environment controller (see Figure 6).

Multithreading is simulated after all simulator events at the current timestamp have been processed (with all reactive interactions between Grid nodes completed). At this point the environment controller is activated. In turn, it activates the scheduler and timer manager of every peer in arbitrary order. The environment controller reads - and resets - its state for every peer: When activation signals have been stored for a given peer, the corresponding scheduler is activated. The timer manager is systematically activated for each peer every simulated time unit.

The environment controller is activated after the processing of simulator events with the same timestamp. This is done to guarantee the completion of all interactions between Grid nodes that result from events with that timestamp. The environment controller could be activated after the processing of each simulator event, but it would probably bring limited benefits. The environment controller could also be activated during the processing of each simulator event. This would require simulator-level multithreading, with a number of threads proportional to the maximum number of simulated peers that could be involved in cascade interactions.

3.3 Resource Virtualization

The task management code is shared by the middleware and simulator implementation of a resource. Task execution and preemption are implemented differently. The limited multithreading within a resource is not simulated because it is not needed. Transfer and storage of input data files required by tasks are currently not simulated.

3.3.1 Simulation of Task Execution

In the middleware implementation, running a task consists of launching within a helper thread an execution module that asynchronously starts and controls a new Java VM to run the task. This is done in order not to block the resource service thread.

In the simulator implementation, running a task essentially consists of deciding whether to simulate a failure of execution or do nothing for some time, i.e. simulate task completion. In the simulator, simulated user agents submit tasks with a fictive run-time (for a resource with a power of one unit), defined by the human user of the simulator. A computing power is associated to each simulated resource by the human user of the simulator. It is therefore straightforward to estimate the simulated run-time of a given task on a given resource. Finally, a simulated resource uploads a dummy output data file to its owner peer when it successfully completes a simulated task.

3.3.2 Simulation of Task Execution Failure

A simulated resource can be configured to fail a small number of randomly selected task executions. This is useful both to test the behavior of peers and the performance of scheduling algorithms. Task execution failures are rare events. They are modelled by a Poissonian process and an exponential distribution is used to determine actual failure probability.

3.3.3 Simulation of Task Preemption

In the middleware implementation, preempting a task consists of asking the execution module to destroy the Java VM where the Grid application is running.

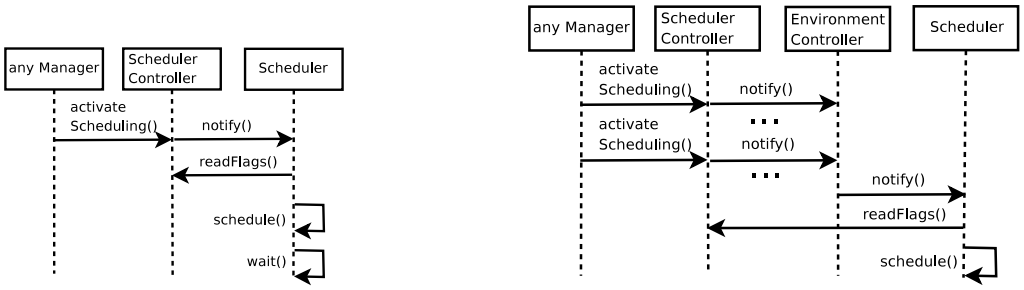
In the simulator implementation, preempting a task is not as straightforward. The state of the simulated resource must be updated in case of simulated task execution failure, which may happen for several reasons: Resource failure, preemption or cancellation (= preemption without subsequent requeueing). Task execution time-out and reclaiming of computational power are the two major causes of preemption/cancellation.

4. SIMULATOR IMPLEMENTATION

The implementation of major simulator components (simulator clock, event list/processor, main simulator loop and simulation description language) are now described.

4.1 Time Management

The middleware and the simulator implementations use the same interface to read the current time. In the middleware implementation, the time is provided by the Java VM and is updated by the computer clock. In the simulator



(a) Scheduler activation (middleware implementation) (b) Scheduler activation (simulator implementation)

Figure 6: Scheduler activation.

implementation, the returned time does not come from the computer clock. The returned time is read from a simulator-wide simulated clock instead. The clock is initialized to zero simulated time units at the beginning of a simulation. It is updated by the event processor only, when all simulator events happening at a given timestamp have been processed. The temporal resolution of the simulated clock, i.e. the value symbolized by one simulated time unit, is currently one second.

In the middleware implementation, time desynchronization between clocks of Grid nodes does not give rise to major issues. Indeed, a P2P Grid is designed to operate in a fully decentralized way. In the simulator implementation, time synchronization between simulated clocks of Grid nodes must be enforced so that the simulator can correctly compute simulation statistics. As simulated Grid nodes transparently share the same simulated clock, continuous time synchronization is guaranteed.

4.2 Simulator Events

Each simulator event represents an asynchronous change in system state and is associated with a timestamp (see Section 2.3). Simulator events are inserted into the event list by simulated Grid nodes and processed in the main simulator loop by the event processor. There are currently four types of simulator events: (1) BoT submission, (2) completion of task execution, (3) failure of task execution, (4) timer event.

The event processor extracts events from the event list (see Figure 7). For each event, the event processor calls code that, in the middleware implementation, would be called from a Grid node following a signal from its environment, e.g. status of task execution or input from a human user.

4.2.1 Processing of BoT Submission Events

Each simulated user agent is activated by the simulator to submit new BoTs to the peer it is using. When submitting a BoT, it simultaneously inserts a new event into the event list so that, at the expected timestamp, the simulator activates it. A new BoT is then submitted and this cycle goes on (until the configured number of BoTs has been submitted). Inserting a submitted BoT event into the event list can be seen as a form of callback mechanism.

4.2.2 Processing of Task Completion/Failure Events

Simulating a simulator event related to task execution is a two-step process. Firstly, when a task is sent to a resource for execution, the simulated resource makes a random decision about its reliability and inserts (with high probability) a completed Task execution event or (with low probability) a failed Task execution event into the event list.

Secondly, when a task execution event occurs, a simulated resource is activated by the simulator. In the middleware implementation, this would happen when the Grid application run by the resource actually completes or fails its execution. Upon a task execution event, the simulated resource - activated by the event processor - either uploads results to its owner peer or notifies that the running task has been preempted. In either case, the peer state is correctly updated.

4.2.3 Processing of Timer Events

The Concurrent execution of the timer manager of each peer is simulated by activating it every simulated time unit, so as to simulate the timer manager thread. The activation of timer managers of all peers is triggered by one simulator timer event. After an event has been processed, a new one is immediately inserted into the event list with a timestamp set one simulated time unit later.

There is one exception to the insertion of a new timer event into the event list: When the event list is empty (provided that there is no running timer that would require the processing of a timer event), no new timer event is inserted into the event list. This exception must be enforced so that the main simulator loop does not enter into an infinite cycle which would lead to a situation where a timer event is removed from the event list (thus emptying it) before a new one is immediately added.

The processing of a timer event is done by activating, sequentially within the main simulator thread, the timer manager of each peer. The environment controller is ideally positioned to perform this activation, as it already has access to peer components.

4.3 Main Simulator Loop

The processing of each event updates the state of the simulated Grid. It often leads simulated Grid nodes to exchange GNMP messages with one another. It also often leads to the generation of new simulator events.

The simulation of multithreading activities, e.g. scheduling, is activated by the environment controller (see Section 3.2.2), when all simulator events with the current timestamp have been processed. Multithreading activities are individually activated for a given peer if activation signals have been emitted (see Figure 6) during the processing of simulator events at the current timestamp.

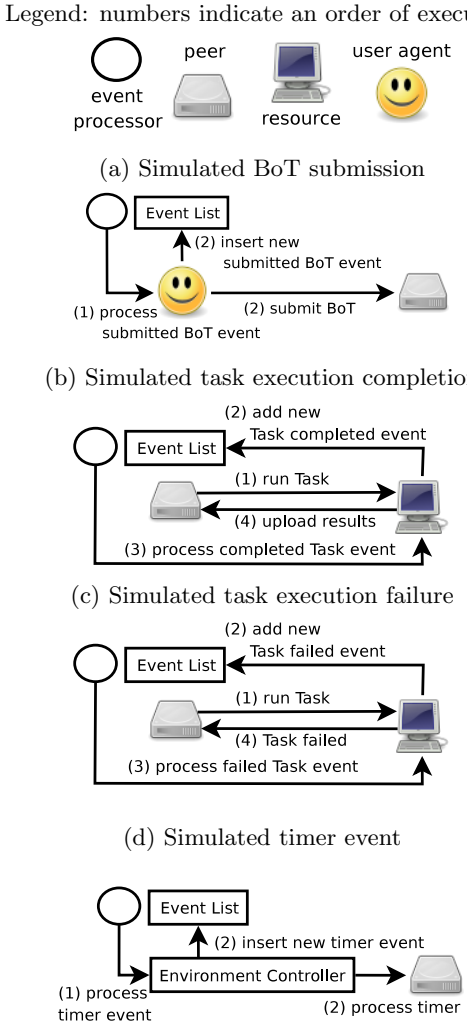


Figure 7: Processing of simulator events.

New simulator events are inserted into the event list as long as simulated user agents submit tasks to peers. The main simulator loop stops when the event list is empty. As submitted tasks complete their simulated execution, simulator events are eventually removed from the event list.

If an event extracted from the event list has a timestamp strictly greater than the timestamp of the previously extracted event, the simulator updates the system-wide clock to simulate the advance of time.

4.4 Limits of the Current Implementation

Transfers of input/output data files between Grid nodes are currently not simulated. The time taken to complete simulated BoTs will inevitably be shorter than what it should be, especially for Data-Intensive BoTs. We are aware of this issue which will require considerable amounts of work in order to be fully addressed; related research that would prove useful in this respect includes Casanova, Legrand et al. [11, 21] and Eger et al. [16].

The activities of multiple peers, notably scheduling, are sequentialized in the simulator implementation because Grid nodes are no longer running independently from one another. GNMP messages (see Section 2.1.3) sent concurrently by live Grid nodes running the middleware implementation are also sequentialized. The processing order of GNMP messages sent simultaneously by multiple Grid nodes will thus vary from what it would be for live Grid nodes running the middleware implementation. Indeed, the order of multithreading simulation as well as the order of insertion and extraction of simultaneous simulator events are currently both arbitrary. Moreover, multithreading simulation takes place only after all simulator events at the current timestamp have been processed.

The processing order of single GNMP messages will also vary slightly from what it would be for live Grid nodes running the middleware implementation. Indeed, small hardware- or network-level variations, e.g. high CPU load or network delays, would be exceedingly difficult to reproduce.

Simulating the network transfers of GNMP messages and introducing true parallelism in the simulation of multithreading would enable the detection of complex timing issues. This requires further work, starting with an event-driven reimplementation of communications within threads of single Grid nodes.

4.5 Simulation Description Language

The configuration of a Grid can be easily described in a simulation description file, also called a *scenario*. A *simulation run* refers to one execution of the simulator with a given scenario, i.e. one simulation of a whole P2P Grid in a controlled environment defined by a given scenario. A simulation description language has been defined (with its own BNF grammar) to easily test various Grid configurations and workloads; only the more relevant parameters are sketched in the next paragraphs.

One user agent is implicitly assigned to each peer (more could be assigned, but our focus is P2P rather than UA2P interactions). A synthetic workload is generated by each user agent according to several parameters: (1) number of BoTs to submit and number of tasks per BoT, (2) initial time shift before beginning to submit BoTs, (3) bounds of the BoT inter-submission time distribution (i.e. time between two consecutive BoT submissions), (4) bounds of the task run-time distribution (for a resource with a power equal to one), (5) description of tasks' input data files. The run-time distribution of simulated tasks is used only in the simulator implementation to simulate task execution and compute execution statistics. This run-time distribution, which would be hard to estimate for real tasks, is not used by the scheduling algorithms.

Each simulated resource is assigned a resource power [26], (in an absolute unit) which is a multiple of a known base power. It is thus easy to set the power of several resources relatively to one another. The peer power represents the total power of the resources managed by the peer; it can be split explicitly or randomly between these resources. An MTBF value (see Section 3.3) can also be specified (independently for each resource) to simulate task execution failures caused by resource failures.

Scheduling policies and several control parameters have to be defined for each simulated peer. In all, 32 parameters (related to scheduling, negotiation, accounting, data management, simulation control) currently have to be defined for each scenario.

5. DISTRIBUTED TESTING

The LBG simulator is not designed to be a distributed application itself, but a set of simulator instances can constitute a distributed application. A distributed testing process is introduced. Varying scheduling policies with a fixed workload can be quickly tested. Simulation results for a fixed scheduling policy can also be smoothed, through the averaging of the results of multiple simulation runs following this policy but with slightly varying workloads.

5.1 Policy Enumerator

The scheduling model presented in Section 2.1.2 is composed of 7 policy decision points. Several policies are implemented for each of them. Testing all valid combinations of scheduling policies before major releases of the middleware implementation is of high interest in practice. Comparing their performance for a given workload is also of high interest. Running these test cases on a Grid enables their completion in a reasonable amount of time, as recently proposed by Duarte et al. [14].

To test as well as evaluate the performance of scheduling policies, we propose to run a large number of instances of the LBG simulator on a Grid based on the LBG middleware. The simulator instances are configured with scheduling policies that vary but with a common workload.

With the evolution of the middleware, the number of test cases will grow to a huge number of valid combinations. For example, there are currently 1446 valid combinations of scheduling policies in LBG (in addition, there are also $\sim$ 30 parameters to configure LBG). Skoll [23] is a distributed continuous testing process. To handle extremely large sets of combinations of policies, it adaptively selects which test cases to actually run. A limited number of test cases are initially selected. Some of them complete their execution, others fail in the sense that they exhibit unexpected behaviors, such as exceptions, run-times errors and assertion failures. Such failures are due either to programming mistakes, i.e. bugs, or to intentional checks manually coded by the Grid application developer, i.e. to detect inconsistencies in data structures or to enforce desirable properties such as system liveness. Each test case that fails leads to the testing of neighbor cases. This tends to minimize the number of test cases to run, while maximizing the coverage of the test case space. Such test case selection techniques can certainly be added to our proposed distributed testing process.

5.2 Scenario Randomizer

The LBG simulator always produces the same output for a given scenario. As a random seed is defined for each simulation run, random variations can be added to the Grid environment, including synthetic workloads and in particular the inter-arrival times of submitted BoTs. Some values of the random seed may lead to "limit cases" of input values that bias the performance of scheduling policies.

To address this issue, we propose to run a large number of instances of the simulator implementation of LBG on a Grid based on its middleware implementation. The simulator instances are configured with the same scheduling policy but with slightly varying workloads, i.e. with a random set of seed values. Simulation results accumulated in successive simulation runs (such as peers utilization and mean BoT response times) are averaged, reducing the influence of the "limit cases" and hopefully smoothing out outlier results.

5.3 Distributed Simulation

Structuring multiple instances of the LBG simulator as a BoT is actually straightforward. A task can be defined for each intended simulation run to be completed. In the following, we refer to such a task as a SimTask, which is an instance of the LBG simulator that is run on the P2P Grid. We also call BoS a Bag of SimTasks. A different scenario is given to each SimTask as an input data file. The output generated by the SimTasks can easily be compared (for the policy enumerator use case) or averaged (for the scenario randomizer use case). Figure 8 illustrates a Bag of 3 SimTasks run on a P2P Grid, each of them simulating a whole P2P Grid.

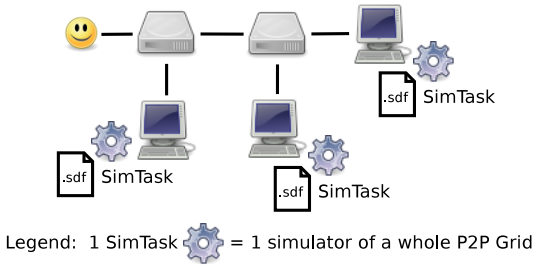


Figure 8: Bag of SimTasks run on a P2P Grid.

Self-bootstrapping is the pattern where a current, stable version of a given system is used to develop the next version of this system. This pattern is common for compilers. Running multiple instances of the LBG simulator on the Grid based on the LBG middleware can be considered as self-bootstrapping. A Grid deployed with basic (or known to be reliable) scheduling policies can help test and evaluate new or improved scheduling policies running in SimTasks. Self-bootstrapping thus enables our research work to contribute to its own evolution.

A limited form of self-bootstrapping has been used for another related P2P Grid middleware, OurGrid [26, 12]. Advanced accounting policies have been simulated by a special-purpose discrete-event system simulator partially based on code from the OurGrid middleware. Multiple instances of this simulator have been structured as a BoS. They were run on a Grid based on the OurGrid middleware deployed with basic accounting policies.

6. EVALUATION

6.1 Co-Development

The initial middleware implementation has been developed in a separate development branch, in parallel with the simulator implementation and scheduling code (which is common to both the middleware and simulator). Following the *code once, deploy twice* pattern has demonstrated several benefits.

Firstly, the separation of concerns has enabled the introduction of well-designed interfaces, simplified the development, and led to a more focused and easier-to-maintain middleware implementation.

Secondly, every bug in the scheduling code was isolated in the controlled environment of the simulator before they appear (or sometimes immediately after) in the middleware

implementation. When an error or unexpected exception occurred when executing the middleware, we used the simulator with identically configured Grid nodes to trace the cause of the problem. The difficulty of testing code on a real network of live Grid nodes has thus been completely avoided, except for a handful of deadlocks and timing issues. These were identified with a run-time monitoring tool[2] providing the state and stack of all threads of target Grid nodes.

Thirdly, deploying new scheduling policies is immediate once they have been tested in the simulator: This speeds up the development cycle and encourages to evaluate new ideas.

6.2 Policy Enumerator

An example of the proposed distributed testing process is now given for the policy enumerator. The introduction of an adaptive preemption policy in the LBG middleware, combined with general purpose refactoring, was done over the course of one week. It brought the code base from 285 source files (391 classes, 53780 lines) to 292 source files (402 classes, 55621 lines), 73 of which were modified (either created or updated).

We then run a BoS of 2892 SimTasks with a scenario involving 20 synthetic Jobs submitted to each of the 15 simulated peers. The BoS was completed in 53 minutes on 100 resources: 2859 SimTasks were successfully completed, 40 SimTasks failed due to an exception in the task control code and 3 SimTasks failed due to an exception in the RMS code. After investigating the stack traces available in the execution logs of the failed SimTasks, we patched the issue in the task control code.

Of the 2892 SimTasks, the configuration of 482 involved the newly implemented adaptive preemption policy, yet only 43 failed due to the task control bug. Without the distributed testing process, this subtle issue would probably not have been resolved before it resurfaced unexpectedly a little while later at a more inconvenient time. Distributed testing certifies that the software behaves as intended for a specific set of well-known, typical Grid configurations and workloads. Complementary mechanisms should also be used but we believe that distributed testing is a valuable tool in a software engineer's toolbox.

6.3 Scenario Randomizer

To smooth simulation results as proposed in Section 5, the simulator can be run multiple times with the same simulation description, except for the master random seed which is different each time. These multiple simulation runs can be structured as a BoS (Bag of SimTasks) that is run by the LBG middleware.

Two scenarios are considered and run multiple times. The first scenario consists of a Grid of 4 peers that manage 4 resources each. Each peer must process 60 (simulated) Bags of 40 Tasks. Peers can barter computing time. They are configured to minimize queueing of external tasks while busy. They also are configured to cancel (rather than preempt) externals tasks when they need to reclaim their resources for their own tasks. The Grid topology is identical in the second scenario, but peers do not barter computing time.

Table 1 shows, for varying numbers of simulation runs, the mean run-time of one simulation and the mean BoT response time (MBRT) of all simulated BoTs. The mean run-time of one simulation is expressed in wall clock time while the MBRT is expressed in simulated time. This experiment shows that a single simulation run often leads to results faster than the average of as few as 10 simulation runs. These results hold for scenarios larger than the considered examples. Extensive statistical studies could be conducted to determine the optimal number of simulation runs required to achieve a given level of confidence. Orthogonally, these results also show that bartering decreases the MBRT.

(a) 4-peers scenario with bartering

# sim runs	1	10	20	40	80	160
sim time (s)	7	9	10	11	8	9
MBRT (s)	767	809	817	832	858	879

(b) 4-peers scenario without bartering

# sim runs	1	10	20	40	80	160
sim time (s)	5	8	8	7	8	9
MBRT (s)	1249	1436	1449	1429	1416	1415

Table 1: Simulation run-times and mean BoT response times achieved with the scenario randomizer.

6.4 Simulator Performance

The performance of the simulator is evaluated according to the simulator run-time, i.e. wall clock time expressed in seconds. The base scenario is the first scenario described in the previous section (4 peers that manage 4 resources each, each peer must process 60 (simulated) Bags of 40 Tasks and peers can barter computing time). The simulator runs presented in Table 2 vary according to one of three parameters: number of submitted BoTs, number of peers in the Grid, number of resources per peer. All simulator runs have been performed on a 64bits Intel Xeon CPU, using 15GB RAM.

Parameter		Run-time (s)
Base scenario		7
BoTs	× 10	10
BoTs	× 100	163
BoTs	× 1000	1605
BoTs	× 10000	18089
peers	× 10	106
peers	× 100	1085
peers	× 200	3457
peers	× 400	7963
peers	× 800	23271
resources	× 10	6
resources	× 100	7
resources	× 1000	11
resources	× 10000	69
resources	× 100000	611

Table 2: Simulator completion run-times with varying scenario parameters.

The simulator performance is linear with the number of submitted BoTs. The 100× BoTs experiment - which involves nearly 1 million tasks (2400 tasks submitted to 4

[2] `http://www.yourkit.com/`

peers, multiplied by 100) - exhibits performance that is asymptotically of the same order of magnitude as the performance achieved by SimGrid [11].

The simulator performance is more than linear with the number of peers (it must be noted that the total number of BoTs submitted in the Grid grows with the number of peers). However, memory consumption becomes unbearable beyond a few thousands peers, precluding larger simulations. Indeed, LBG peers memorize a lot of data about their interactions with other peers (see Section 2.1.1): Even if the interaction history is strictly bounded, the memory requirements of the simulator are quadratic with the number of simulated peers. Memory management can be optimized by both exploiting secondary storage and tuning the peer discovery process. We believe that controlling memory consumption can dramatically improve the simulator performance in scenarios involving large numbers of peers.

The simulator performance is less than linear with the number of resources per peer, as expected. Such scenarios lead to less simulator iterations as the increased computational power enables peers to process BoTs much faster. Nonetheless, the management of several hundred thousands resources per peer induces a small performance penalty on the simulation (but negligible for individual Grid nodes).

6.5 Simulator Bias

The time spent by the peers scheduling code is neglected in the time management process. This small, yet systematic, bias in time simulation is however acceptable. There are many orders of magnitude between the time taken by peers to process an incoming GNMP message (a few milliseconds) and the time taken to execute a task on a resource (several minutes to several hours).

On a typical desktop computer, a test scenario may take less than 1 minute of wall clock time to simulate about 3 hours of operations of a medium-sized Grid (15 peers, 500 resources). Let $r_s = 3 \times 3600$ seconds $= 10800$ seconds be the simulated run-time of a given scenario. Let $r_p = (60$ seconds / 15 peers$) = 4$ seconds be the average run-time of peer code (which is identical in both the middleware and simulator implementations, thanks to massive code reuse).

For this typical test scenario, the time bias is bounded by much less than 1% of the simulated execution time ($r_p/r_s = 0.04\%$). Furthermore, this represents an overly large upper bound on simulation bias, considering that some wall clock time is spent by the simulator to initialize its data structures, manage the event list and the environment controller.

7. RELATED WORK

There exists few Grid simulators: Bricks [2], ChicSim [25], GangSim [15], GridSim [10], GSSim [20], OurGrid simulator [26], SimGrid [11].

The discrete-event system simulator most closely related to LBG is SimGrid [11]. It is a very advanced and flexible Grid simulator and middleware targeting the evaluation of distributed systems. It is the only existing Grid simulator (besides ours, and to the best of our knowledge) to support the *code once, deploy twice* pattern. Code reuse is an important design goal of SimGrid. Through its GRAS component [24], it exposes an API of low level primitives suitable for communications in applicative overlay applications. It allows developers to easily simulate, and also run as part of a middleware, code that is built on top of SimGrid components. GRAS requires to adhere to its API when developing scheduling algorithms. The LBG simulator, on the other hand, is tailored to the middleware (although the approach itself is generic). GRAS would correspond to GNMP services and handles (see Section 2.1.3) packaged into an API, augmented with timing management (also available in our simulator) but without support for simulation of multithreaded code. LBG could be seen as a synthesis of variants of the OurGrid [12] middleware and the SimGrid [11, 24] middleware/simulator.

GSSim [20], built on top of GridSim [10], is another closely related discrete-event Grid simulator. It shares with SimGrid and our simulator the goal to virtualize a Grid in order to use the same code both in the simulator and the middleware code. Like SimGrid, it targets the evaluation of scheduling algorithms. However, code reuse is limited to the scheduling algorithms. A related web portal[3] acts as a repository of trace workloads and scheduling algorithms.

8. CONCLUDING REMARKS

8.1 Summary of Contributions

Coding once and deploying twice enables reproducible testing of distributed software. A middleware implementation can be virtualized so that a simulator can be embedded: Only communications between Grid nodes, multithreading within Grid nodes and actual task execution are coded differently in the simulator implementation. The middleware and the simulator consequently share most of their code. This shared code, which includes the scheduling code, can be tested within the controlled environment of the simulator. Distributed testing helps speed up the testing process. The shared code can subsequently be deployed without any modification as part of the middleware, on real networked computers. This promotes separation of concerns, as well as easy prototyping, evaluation and integration of new scheduling algorithms.

8.2 Future Work

The simulation of multithreading activities should be extended and more flexible; recent research on run-time monitoring [1] and testing [13] of multi-threaded code may provide insights in this respect.

Data transfers are currently not taken into account. Simulating data transfers will considerably improve simulation accuracy when dealing with Data-Intensive BoTs [8].

It would be useful to define an API to facilitate the use of the *code once, deploy twice* pattern, as is done in SimGrid [11, 24]. It would also be useful to standardize the description of Grid configurations (including which scheduling algorithms to use) and deployment metadata, as is also done in SimGrid [11] and in GSSim [20]. Adding support for trace workloads would also be of high interest.

Acknowledgments

We want to thank Xavier Dalem for code contributed to the LBG middleware; Claire Kopacz for her help in the preparation of this manuscript; and the anonymous reviewers for their constructive comments. Some figures include icons from the Tango library (`http://tango.freedesktop.org/`), under Creative Commons Attribution Share-Alike license.

[3] `http://www.gssim.org/`

9. REFERENCES

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static analysis and Run-time Monitoring. In *Proc. PADTAD, Verification Conference Workshops*, Haifa, Israel, November 2005.

[2] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. . Sekiguchi, and U. Nagashima. Performance Evaluation Model for Scheduling in Global Computing Systems. In *Int. J. High Performance Computing Applications*, volume 14, pages 268–279. SAGE, 2000.

[3] K. Amin, G. von Laszewski, and A. Mikler. Grid Computing for the Masses: An Overview. In *Proc. Workshop on Grid and Cooperative Computing*, Shangai, China, 2003.

[4] D. Anderson and G. Fedak. The Computational and Storage Potential of Volunteer Computing. In *Proc. CCGRID 2006*, Singapore, 2006.

[5] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proc. Grid*, Pittsburgh, PA, USA, November 2004.

[6] J. Banks, J. Carson, B. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 3rd edition, 2000.

[7] F. Brasileiro, E. Araújo, W. Voorsluys, M. Oliveira, and F. Figueiredo. Bridging the High Performance Computing Gap: the OurGrid Experience. In *Proc. LAGrid, CCGrid Workshops*, Rio de Janeiro, Brazil, May 2007.

[8] C. Briquet, X. Dalem, S. Jodogne, and P.-A. de Marneffe. Scheduling Data-Intensive Bags of Tasks in P2P Grids with BitTorrent-enabled Data Distribution. In *Proc. UPGRADE-CN'07, HPDC Workshops*, Monterey Bay, CA, USA, June 2007.

[9] C. Briquet and P.-A. de Marneffe. Description of a Lightweight Bartering Grid Architecture. In *Proc. Cracow Grid Workshop*, Cracow, Poland, 2006.

[10] R. Buyya and M. Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. In *Journal of Concurrency and Computation: Practice and Experience*. Wiley Press, USA, 2002.

[11] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experimentations. In *Proc. UKSim*, Cambridge, UK, April 2008.

[12] W. Cirne, F. Brasileiro, N. Andrade, L. B. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! In *J. Grid Computing*. Springer, 2006.

[13] A. Dantas, W. Cirne, and F. Brasileiro. Improving Automated Testing of Multi-threaded Software. In *Proc. Int. Conf. Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008.

[14] A. Duarte, F. Wagner, Gustavo Brasileiro, and W. Cirne. Multi-Environment Software Testing on the Grid. In *Proc. PADTAD, ISSTA Workshops*, London, UK, July 2006.

[15] C. Dumitrescu and I. Foster. GangSim: A Simulator for Grid Scheduling Studies. In *Proc. CCGrid*, Cardiff, UK, 2005.

[16] K. Eger, T. Hoßfeld, A. Binzenhöfer, and G. Kunzmann. Simulation of Large-Scale P2P Networks: Packet-level vs. Flow-level Simulations. In *Proc. UPGRADE-CN'07, HPDC Workshops*, 2007.

[17] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. Supercomputer App.*, 15(3), 2001.

[18] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *SIGPLAN Notices*, volume 39. ACM, December 2004.

[19] P. H. Jacobs and A. Verbraeck. Single-Threaded Specification of Process-Interaction Formalism in Java. In *Proc. Winter Simulation Conference*, Washington, DC, USA, 2004.

[20] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz. Grid Scheduling Simulations with GSSIM. In *Proc. 3rd International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, Hsinchu, Taiwan, December 2007.

[21] A. Legrand, M. Quinson, K. Fujiwara, and H. Casanova. The SimGrid Project - Simulation and Deployment of Distributed Applications. In *Proc. HPDC*, Paris, France, May 2006.

[22] R. Olejnik, B. Toursel, M. Tudruj, and E. Laskowski. DG-ADAJ: a Java Computing Platform for Desktop Grid. In *Proc. Cracow Grid Workshop*, Cracow, Poland, 2005.

[23] A. Porter, A. Memon, C. Yilmaz, D. C. Schmidt, and B. Natarajan. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. In *IEEE Transactions on Software Engineering*, volume 33. 2007.

[24] M. Quinson. GRAS: a Research and Development Framework for Grid and P2P Infrastructures. In *Proc. Parallel and Distributed Computing and Systems*, Dallas, TX, USA, November 2006.

[25] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proc. HPDC*, Edinburgh, Scotland, UK, 2002.

[26] R. Santos, A. Andrade, W. Cirne, F. Brasileiro, and N. Andrade. Relative Autonomous Accounting for Peer-to-Peer Grids. In *Concurrency and Computation: Practice and Experience*, volume 19. Wiley, 2007.

[27] Wikipedia, the Free Encyclopedia.