Reproducible Testing of Distributed Software with Middleware Virtualization and Simulation

Cyril Briquet
Department of EE & CS
University of Liège, Belgium
P2P Grid middleware developed at University of Liege, BE:

**Lightweight Bartering Grid**

- study *distributed scheduling* policies in unreliable environments
- provide *software engineering tools* for *testing* and performance evaluation
Goal: testing Grid middleware

- **reproducible testing** of most of the middleware code
- **accurate** discrete-event **simulation** of a whole Grid
How: abstraction of Grid nodes

Communications between Grid nodes, as well as timers, multithreading...
(interactions with system APIs)

can be abstracted, then simulated

A whole P2P Grid can then run

within a discrete-event simulator
Contents

- Introduction
- **Context**
- Virtualization and simulation
- Reproducible testing
- Summary
P2P Grid topology: 2-levels

- worker node
  (network-enabled edge computer)

- Peer = controller of worker nodes, server, consumer and supplier of computational power
2-step Task submission

informational opacity between Peers
Features of a P2P Grid (1/2)

- P2P = bottom-up Grid formation (Peers freely leave and join)
- nondedicated Grid nodes
- worker nodes can register/deregister dynamically (1 Peer of a P2P Grid ~ Volunteer Grid, e.g. SETI@home)
- (most) interactions can be asynchronous
• submitted Tasks can be cancelled – at any time - by user agents, worker nodes and also Peers (=> bursts of preemption)

• indeed, Peers can reclaim at any time for their own use their computational power supplied to other Peers
Contents

- Introduction
- Context
- Virtualization and simulation
- Reproducible testing
- Summary
Testing is difficult

P2P Grid middleware

- large software (for Peers, worker nodes, user agents)
- multiple sources of bugs
  (scheduling algorithms, state consistency, network, code execution, multithreading, data transfers)
Testing is difficult (2/2)

- unit testing is nice, but until P2P Grid middlewares can be fully verified, integration tests for typical setups are necessary

- **unit testing not sufficient, because of interactions** (nontrivial impact of scheduling policies on one another)

- big issue: difficult to reproduce a given P2P Grid state (P2P Grid = complex distributed environment)
Performance evaluation is difficult

- scheduling algorithms: mostly heuristics
  (analytical models too complex)

  => performance evaluated experimentally

- big issue: difficult to reproduce a given P2P Grid state
  (P2P Grid = complex distributed environment)
Discrete-event system simulation

Purpose:
- observe a system in a controlled environment

Benefits:
- implementing a simulator easier than setting up a large-scale testbed
- simulations totally controllable, repeatable
- fast (vs. real execution, or emulation)

Accuracy?
Virtualization and simulation (1/2)

- our contribution:
  use virtualization, simulation as software engineering tools

- virtualization = injection of an abstraction layer between « layer n » and « layer n-1 »;
in our context: middleware and calls to system APIs
Virtualization and simulation (2/2)

- virtualization of middleware (Virtual Machine, O.S. level)
  i.e. Grid nodes isolated from their environment,
  i.e. scheduling code (+ ... ) uses an abstraction layer

- integration of scheduling code and simulator code:

  Grid nodes are loaded in the simulator memory space,
  each with its own data structures,
  then interact as they would on real computers
Consequences of virtualization, sim

- Grid nodes not aware whether they run on real computers or within the simulator

- massive code reuse between implementations
  (« code once, deploy twice »)

- identical (not simplified) algorithms and architecture between middleware and simulator
  - new policies automatically deployed in middleware
  - high accuracy of performance evaluation
Example: simple scheduling event

(1) run Task

(2) add new Task completed event

(3) process completed Task event

(4) upload results
Simulator events

- workload submission
- (completed, failed) Task execution
- timers
- multithreading (limited model)
- data transfers
  (doable, but requires fine-grained virtualization of data transfer software, i.e. FTP and BitTorrent)
Messaging in the middleware
Messaging in the simulator
Simulator I/O

Inputs:

• topology & computational power of Grid nodes
• configuration of Peer scheduling policies
• description of synthetic workload

Outputs:

• execution statistics (mean BoT response time, utilization)
Contents

- Introduction
- Context
- Virtualization and simulation
- Reproducible testing
- Summary
Use #1: testing before deployment

- unit testing a new scheduling policy:
  - run a P2P Grid in the simulator with a setup to activate some specific code chunk
  - observe outcome, i.e. crashes? assertions not met?
Use #2: testing after deployment

- when an issue arises with middleware (real deployment)
- P2P Grid with same configuration is run in the simulator

=>

determine whether a bug, crash, ...
comes from the common code,
or may come from code interacting with environment

- enables co-development
  - main branch: architecture + simulator
  - secondary branch: Task execution, network
Use #3: policy enumerator (1/3)

• example: introduction of an adaptive preemption policy
  391 classes, 53.8kloc => 402 classes, 55.6kloc,
  + 73 classes refactored

• nothing broken by refactoring?

• unit testing not sufficient, because of interactions
  (nontrivial impact of scheduling policies on one another)
Peer =
server, consumer and supplier of computational power

Legend:
1. Local Tasks scheduling
2. Consumption Tasks scheduling
3. Supplying Tasks filtering
4. Supplying Tasks scheduling
5. Supplying Tasks preemption

outbound Tasks
inbound Tasks
Use #3: policy enumerator (2/3)

- each Peer:
  5 scheduling PDPs, 2 other PDPs, 30+ parameters

- let's enumerate
  2892 combinations of policies = 2892 simulations

- outcome: 2859 OK, 40 bug #1, 3 bug #2
Use #3: policy enumerator (3/3)

- **is it tractable?** today's situation:
  508 classes, 70.5kloc, 16850 combinations and growing

  => all simulations run... on a Grid

  (a few hours to a few days, depending on Grid topology)

- **future:** insights from **Skoll?**
  distributed continuous testing project,
  using **adaptive selection of test cases,**
  with additional tests performed
  in the « neighborhood » of failed test cases
Self-Bootstrapping

• self-bootstrapping = current, stable version of a given system used to develop next version

• 1 middleware:
  basic policies

• N simulators:
  test and evaluate advanced policies
Limits of virtualization & sim

- suitable for long-running Tasks, e.g. runtime > k seconds

- simulation of multithreading should be improved

- simulator is (currently) single-threaded; could benefit from future many-core computers

- transfer of control messages simulated, but transfer of data files (currently) not simulated
Implementation remarks

- **event-driven** architecture and asynchronous messages particularly suitable to discrete-event simulation
- **Java** (J2SE 5.0) middleware, simulator & applications
- OO paradigm very useful:
  - **interfaces with 2 implementations** (middleware, simulator)
- **automated memory management** very useful
  (in the future, reuse of discarded data structures between simulated Grid nodes would be helpful)
Contents

• Introduction
• Context
• Virtualization and simulation
• Reproducible testing
• **Summary**
Summary

- « code once, deploy twice » based on virtualization of Grid nodes at middleware level, and simulation of a whole P2P Grid
- massive code reuse
  => co-development of simulator and middleware
  => immediate deployment of new policies
  => testing before and after deployment
  => systematic testing of policy combinations
  => accurate simulation results
- future work in multithreading, sim of data transfers
Thank You.