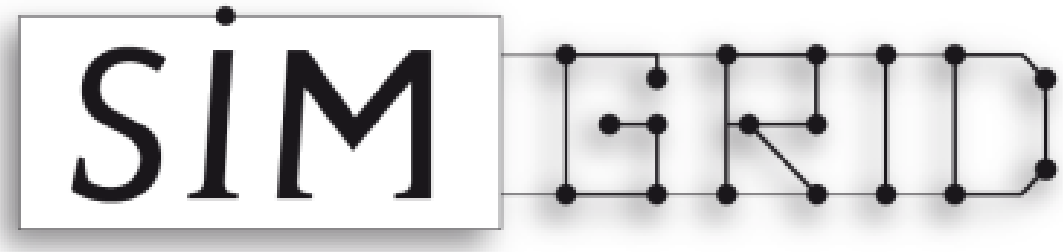
The SimGrid Framework for Research on Large-Scale Distributed Systems

Martin Quinson (Nancy University, France)
Arnaud Legrand (CNRS, Grenoble University, France)
Henri Casanova (Hawai'i University at Manoa, USA)

Presented By:
Pedro Velho (Grenoble University, France)

simgrid-dev@gforge.inria.fr



Large-Scale Distributed Systems Research

Large-scale distributed systems are in production today

- Grid platforms for "e-Science" applications
- Peer-to-peer file sharing
- Distributed volunteer computing
- Distributed gaming

Researchers study a broad range of systems

- Data lookup and caching algorithms
- Application scheduling algorithms
- Resource management and resource sharing strategies

They want to study several aspects of their system performance

- Response time
- Throughput
- Scalability
- Robustness
- Fault-tolerance
- Fairness

Main question: comparing several solutions in relevant settings

Large-Scale Distributed Systems Science?

Requirement for a Scientific Approach

- Reproducible results
  - You can read a paper,
  - reproduce a subset of its results,
  - improve
- Standard methodologies and tools
  - Grad students can learn their use and become operational quickly
  - Experimental scenario can be compared accurately

Current practice in the field: quite different

- Very little common methodologies and tools
- Experimental settings rarely detailed enough in literature (test source codes?)

Purpose of this workshop

- Present "emerging" methodologies and tools
- Show how to use some of them in practice
- Discuss open questions and future directions

Agenda

- Distributed Systems Experiments
  - Methodological Issues
  - Main Methodological Approaches
  - Tools for Experimentations in Large-Scale Distributed Systems
- Resource Models
  - Analytic Models Underlying SimGrid
- SimGrid Architecture and Features
  - Overview of the SimGrid Components
  - SimDag: Comparing Scheduling Heuristics for DAGs
  - MSG: Comparing Heuristics for Concurrent Sequential Processes
  - GRAS: Developing and Debugging Real Applications
  - SMPI: Running MPI applications on top of SimGrid
- Conclusion

Agenda

Analytical or Experimental?

Analytical works?

- Some purely mathematical models exist
- ☺ Allow better understanding of principles in spite of dubious applicability
  impossibility theorems, parameter influence, ...
- ☹ Theoretical results are difficult to achieve
  - Everyday practical issues (routing, scheduling) become NP-hard problems
    Most of the time, only heuristics whose performance have to be assessed are proposed
  - Models too simplistic, rely on ultimately unrealistic assumptions.

$\Rightarrow$ One must run experiments

$\rightsquigarrow$ Most published research in the area is experimental

Running real-world experiments

- ☺ Eminently *believable* to demonstrate the proposed approach applicability
- ☹ Very time and labor consuming
  - Entire application must be functional
  - Parameter-sweep; Design alternatives
- ☹ Choosing the right testbed is difficult
  - My own little testbed?
    - ☺ Well-behaved, controlled,stable
    - ☹ Rarely representative of production platforms
  - Real production platforms?
    - Not everyone has access to them; CS experiments are disruptive for users
    - Experimental settings may change drastically during experiment (components fail; other users load resources; administrators change config.)
- ☹ Results remain limited to the testbed
  - Impact of testbed specificities hard to quantify ⇒ collection of testbeds...
  - Extrapolations and explorations of "what if" scenarios difficult (what if the network were different? what if we had a different workload?)
- ☹ Experiments are uncontrolled and unrepeatable

  No way to test alternatives back-to-back (even if disruption is part of the experiment)

Difficult for others to reproduce results

even if this is the basis for scientific advances!

Simulation

☺ Simulation solves these difficulties

- No need to build a real system, nor the full-fledged application
- Ability to conduct controlled and repeatable experiments
- (Almost) no limits to experimental scenarios
- Possible for anybody to reproduce results

Simulation in a nutshell

- *Predict aspects of the behavior of a system using an approximate model of it*
- Model: Set of objects defined by a state $\oplus$ Rules governing the state evolution
- Simulator: Program computing the evolution according to the rules
- Wanted features:
  - Accuracy: Correspondence between simulation and real-world
  - Scalability: Actually usable by computers (fast enough)
  - Tractability: Actually usable by human beings (simple enough to understand)
  - Instanciability: Can actually describe real settings (no magical parameter)

Simulation in Computer Science

Microprocessor Design

- A few standard "cycle-accurate" simulators are used extensively
  `http://www.cs.wisc.edu/~arch/www/tools.html`

⇒ Possible to reproduce simulation results

Networking

- A few established "packet-level" simulators: NS-2, DaSSF, OMNeT++, GTNetS
- Well-known datasets for network topologies
- Well-known generators of synthetic topologies
- SSF standard: `http://www.ssfnet.org/`

⇒ Possible to reproduce simulation results

Large-Scale Distributed Systems?

- No established simulator up until a few years ago
- Most people build their own "ad-hoc" solutions

> Naicken, Stephen *et Al.*, *Towards Yet Another Peer-to-Peer Simulator*, HET-NETs'06.

From 141 P2P sim.papers, 30% use a custom tool, 50% don't report used tool

Simulation in Parallel and Distributed Computing

- Used for decades, but under drastic assumptions in most cases

Simplistic platform model

- Fixed computation and communication rates (Flops, Mb/s)
- Topology either fully connected or bus (no interference or simple ones)
- Communication and computation are perfectly overlappable

Simplistic application model

- All computations are CPU intensive (no disk, no memory, no user)
- Clear-cut communication and computation phases
- Computation times even ignored in Distributed Computing community
- Communication times sometimes ignored in HPC community

Straightforward simulation in most cases

- Fill in a Gantt chart or count messages with a computer rather than by hand
- No need for a "simulation standard"

Large-Scale Distributed Systems Simulations?

Simple models justifiable at small scale

- Cluster computing (matrix multiply application on switched dedicated cluster)
- Small scale distributed systems

Hardly justifiable for Large-Scale Distributed Systems

- Heterogeneity of components (hosts, links)
  - Quantitative: CPU clock, link bandwidth and latency
  - Qualitative: ethernet *vs* myrinet *vs* quadrics; Pentium *vs* Cell *vs* GPU
- Dynamicity
  - Quantitative: resource sharing $\rightsquigarrow$ availability variation
  - Qualitative: resource come and go (churn)
- Complexity
  - Hierarchical systems: grids of clusters of multi-processors being multi-cores
  - Resource sharing: network contention, QoS, batches
  - Multi-hop networks, non-negligible latencies
  - Middleware overhead (or optimizations)
  - Interference of computation and communication (and disk, memory, etc)

Agenda

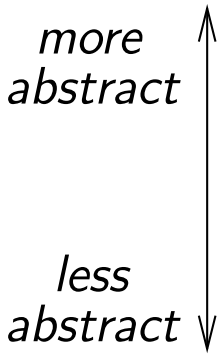
Models of Large-Scale Distributed Systems

Model = Set of objects defined by a state $\oplus$ Set of rules governing the state evolution

Model objects:

- Evaluated application: Do actions, stimulus to the platform
- Resources (network, CPU, disk): Constitute the platform, react to stimulus.
  - Application blocked until actions are done
  - Resource can sometime "do actions" to represent external load

Expressing interaction rules



Mathematical Simulation: Based solely on equations

Discrete-Event Simulation: System = set of dependant actions & events

Emulation: Trapping and virtualization of low-level application/system actions

Real execution: No modification

Boundaries are blurred

- Tools can combine several paradigms for different resources
- Emulators may use a simulator to compute resource availabilities

Simulation options to express rules

Network

- Macroscopic: Flows in "pipes" (mathematical & coarse-grain d.e. simulation)
  Data sizes are "liquid amount", links are "pipes"
- Microscopic: Packet-level simulation (fine-grain d.e. simulation)
- Emulation: Actual flows through "some" network timing + time expansion

CPU

- Macroscopic: Flows of operations in the CPU pipelines
- Microscopic: Cycle-accurate simulation (fine-grain d.e. simulation)
- Emulation: Virtualization via another CPU / Virtual Machine

Applications

- Macroscopic: Application = analytical "flow"
- Less macroscopic: Set of abstract tasks with resource needs and dependencies
  - Coarse-grain d.e. simulation
  - Application specification or pseudo-code API
- Virtualization: Emulation of actual code trapping application generated events

Large-Scale Distributed Systems Simulation Tools

A lot of tools exist

- Grid'5000, Planetlab, MicroGrid, Modelnet, Emulab, DummyNet
- ns-2, GTNetS, SSFNet
- ChicagoSim, GridSim, OptorSim, SimGrid, ...
- PeerSim, P2PSim, ...

How do they compare?

- How do they work?
  - Components taken into account (CPU, network, application)
  - Options used for each component (direct execution; emulation; d.e.; simulation)
- What are their relative qualities?
  - Accuracy (correspondence between simulation and real-world)
  - Technical requirement (programming language, specific hardware)
  - Scale (tractable size of systems at reasonable speed)
  - Experimental settings configurable and repeatable, or not

Experimental tools comparison

	CPU	Disk	Network	Application	Requirement	Settings	Scale
Grid'5000	direct	direct	direct	direct	access	fixed	<5000
Planetlab	virtualize	virtualize	virtualize	virtualize	none	uncontrolled	hundreds
Modelnet	-	-	emulation	emulation	lot material	controlled	dozens
MicroGrid	emulation	-	fine d.e.	emulation	none	controlled	hundreds
ns-2	-	-	fine d.e.	coarse d.e.	C++ and tcl	controlled	<1,000
SSFNet	-	-	fine d.e.	coarse d.e.	Java	controlled	<100,000
GTNetS	-	-	fine d.e.	coarse d.e.	C++	controlled	<177,000
ChicSim	coarse d.e.	-	coarse d.e.	coarse d.e.	C	controlled	few 1,000
OptorSim	coarse d.e.	amount	coarse d.e.	coarse d.e.	Java	controlled	few 1,000
GridSim	coarse d.e.	coarse d.e.	coarse d.e.	coarse d.e.	Java	controlled	few 1,000
P2PSim	-	-	-	state machine	C++	controlled	few 1,000
PlanetSim	-	-	cste time	coarse d.e.	Java	controlled	100,000
PeerSim	-	-	-	state machine	Java	controlled	1,000,000
SimGrid	math/d.e.	(underway)	math/d.e.	d.e./emul	C or Java	controlled	few 100,000

- Direct execution $\rightsquigarrow$ no experimental bias (?)
  Experimental settings fixed (between hardware upgrades), but not controllable
- Virtualization allows sandboxing, but no experimental settings control
- Emulation can have high overheads (but captures the overhead)
- Discrete event simulation is slow, but hopefully accurate
  To scale, you have to trade speed for accuracy

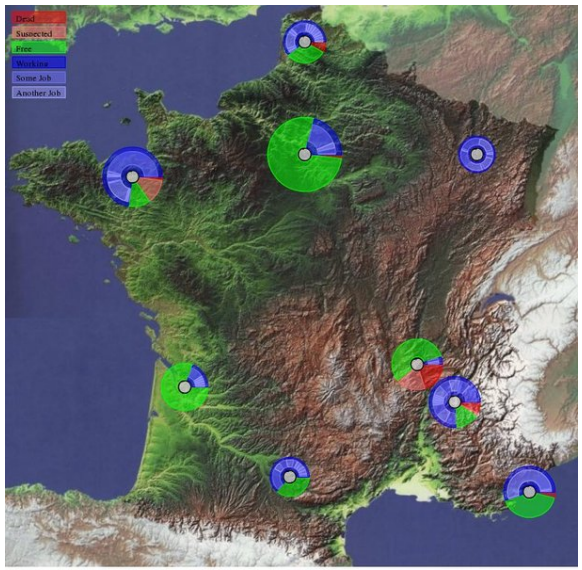
Grid'5000 (consortium – INRIA)

French experimental platform

- 1500 nodes (3000 cpus, 4000 cores) over 9 sites
- Nation-wide 10Gb dedicated interconnection
- `http://www.grid5000.org`

Scientific tool for computer scientists

- Nodes are *deployable*: install your own OS image
- Allow study at any level of the stack:
  - Network (TCP improvements)
  - Middleware (scalability, scheduling, fault-tolerance)
  - Programming (components, code coupling, GridRPC)
  - Applications



☺ Applications not modified, direct execution

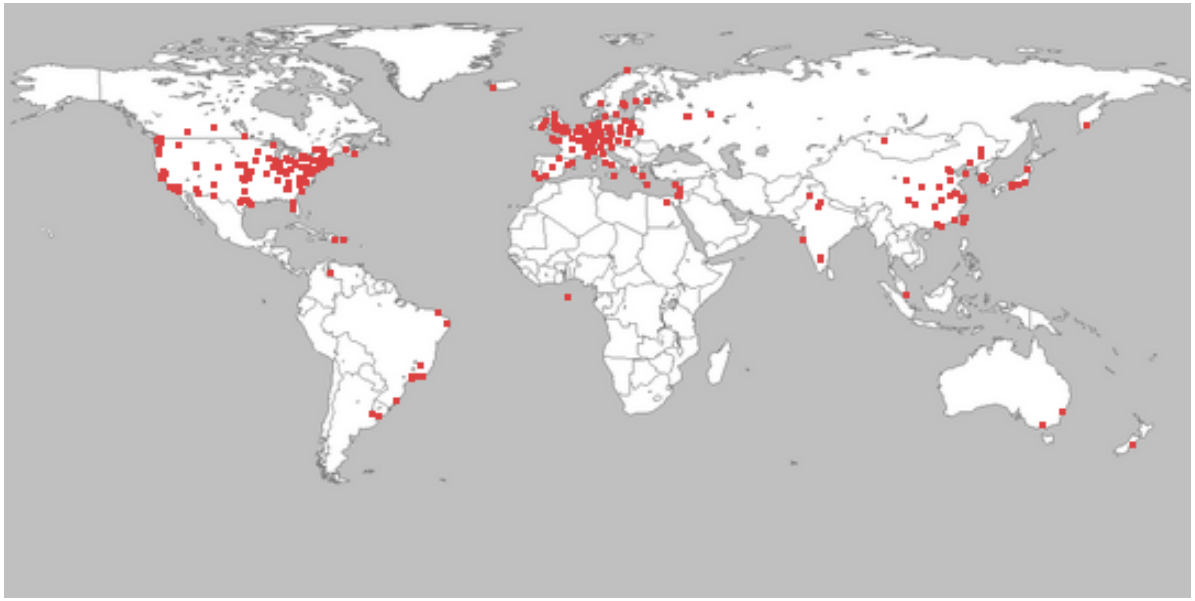
☺ Environment controlled, experiments repeatable

☹ Relative scalability ("only" 1500-4000 nodes)

PlanetLab (consortium)

Open platform for developping, deploying, and accessing planetary-scale services

Planetary-scale 852 nodes, 434 sites, >20 countries



Distribution Virtualization each user can get a slice of the platform

Unbundled Management

- local behavior defined per node; network-wide behavior: services
- multiple competing services in parallel (shared, unprivileged interfaces)

As unstable as the real world

- ☺ Demonstrate the feasability of P2P applications or middlewares
- ☹ No reproducibility!

ModelNet (UCSD/Duke)

Applications

- Emulation and virtualization: Actual code executed on "virtualized" resources
- Key tradeoff: scalability versus accuracy

Resources: system calls intercepted

- `gethostname`, sockets

CPU: direct execution on CPU

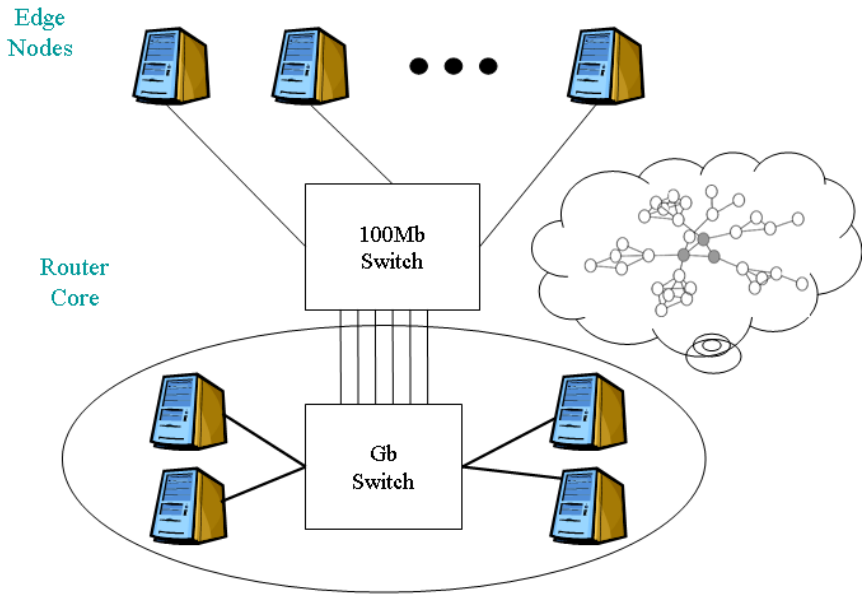
- Slowdown not taken into account!

Network: emulation through:

- one emulator (running on FreeBSD)
- a gigabit LAN
- hosts + IP aliasing for virtual nodes

$\rightsquigarrow$ emulation of heterogeneous links

- Similar ideas used in other projects (Emulab, DummyNet, Panda, ...)

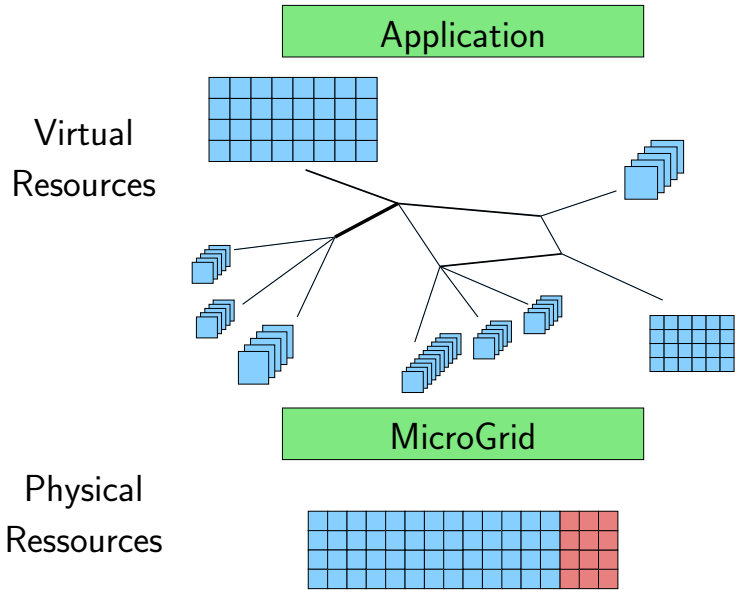


Amin Vahdat *et Al.*, *Scalability and Accuracy in a LargeScale Network Emulator*, OSDI'02.

MicroGrid (UCSD)

Applications

- Application supported by emulation and virtualization
- Actual application code is executed on "virtualized" resources
- Accounts for CPU and network



Resources: wraps syscalls & grid tools

- gethostname, sockets, GIS, MDS, NWS

CPU: direct execution on fraction of CPU

- finds right mapping

Network: packet-level simulation

- parallel version of MaSSF

Time: synchronize real and virtual time

- find the good execution rate

Andrew Chien *et Al.*, *The MicroGrid: a Scientific Tool for Modeling Computational Grids*, Super-Computing 2002.

More recent emulation projects

CPU/OS emulation Lately, there has been lots of efforts on OS emulation / virtualization / para-virtualization / ...
Evolution both on the hardware and OS/software part

- VMWare
- kvm/Qemu
- VirtualBox
- Xen
- VirtualPC
- ...

The effort is made on portability, efficiency, isolation

Network emulation Probably a lot of ongoing projects but my informations are not up-to-date

Two interesting projects developed in the Grid5000 community

- Wrekavoc (Emmanuel Jeannot). Very lightweight and easy to set up (CPU burn, suspend/resume process, tc/IProute2)
- P2PLab (Lucas Nussbaum). An emulation framework specifically designed for the study of P2P systems, where the core of the network is not the bottleneck

Packet-level simulators

ns-2: the most popular one

- Several protocols (TCP, UDP, ...), several queuing models (DropTail, RED, ...)
- Several application models (HTTP, FTP), wired and wireless networks
- Written in C++, configured using TCL. Limitated scalability ($< 1,000$)

SSFNet: implementation of SSF standard

- Scalable Simulation Framework: unified API for d.e. of distributed systems
- Written in Java, usable on 100 000 nodes

GTNetS: Georgia Tech Network Simulator

- Design close to real networks protocol philosophy (layers stacked)
- C++, reported usable with 177,000 nodes

Simulation tools of / for the networking community

- Topic: Study networks behavior, routing protocols, QoS, ...
- Goal: Improve network protocols

$\rightsquigarrow$ Microscopic simulation of packet movements

$\Rightarrow$ Inadequate for us (long simulation time, CPU not taken into account)

Latest news about packet-level simulators

ns-3 is the new promising project

- ns-2 was written in TCL and C++ and was... dirty
  There was many many contributors though and had the largest available protocol code-base
- There is an international effort for rewriting it (cleaner, more efficient, better support for contribution, . . .)

Packet-level simulation and emulation have never been that close

Among the recent interesting features:

- Use real TCP stacks (e.g., from the Linux kernel) instead of TCP models
- The Distributed Client Extension (available in ns-2): one computer (EmuHost) runs ns-2, a set of computers running the real application (RWApps)
  (De)multiplexing and UDP tunneling of traffic between the EmuHost and the RWApps using TAP

ChicagoSim, OptorSim, GridSim, ...

- Network simulator are not adapted, emulation solutions are too heavy
- PhD students just need simulator to plug in their algorithm
  - Data placement/replication
  - Grid economy

$\Rightarrow$ Many simulators. Most are home-made, short-lived; Some are released

ChicSim designed for the study of data replication (Data Grids), built on ParSec
Ranganathan, Foster, *Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications*, HPDC'02.

OptorSim developped for European Data-Grid
DataGrid, CERN. *OptorSim: Simulating data access optimization algorithms*

GridSim focused on Grid economy
Buyya et Al. *GridSim: A Toolkit for the Modeling and Simulation of Global Grids*, CCPE'02.

every [sub-]community seems to have its own simulator

PeerSim, P2PSim, ...

Thee peer-to-peer community also has its own private collection of simulators:

$$\text{focused on P2P protocols} \rightsquigarrow \text{main challenge} = \text{scale}$$

P2PSim Multi-threaded discrete-event simulator. Constant communication time. Alpha release (april 2005)

`http://pdos.csail.mit.edu/p2psim/`

PlanetSim Multi-threaded discrete-event simulator. Constant communication time. Last release (2006)

`http://planet.urv.es/trac/planetsim/wiki/PlanetSim`

PeerSim Designed for epidemic protocols. processes = state machines. Two simulation modes: cycle-based (time is discrete) or event-based. Resources are not modeled. 1.0.3 release (december 2007)

`http://peersim.sourceforge.net/`

OverSim A recent one based on OMNeT++ (april 2008)

`http://www.oversim.org/`

SimGrid (Hawai'i, Grenoble, Nancy)

History

- Created just like other home-made simulators (only a bit earlier ;)
- Original goal: scheduling research $\rightsquigarrow$ need for speed (parameter sweep)
- HPC community concerned by performance $\rightsquigarrow$ accuracy not negligible

SimGrid in a Nutshell

- Simulation $\equiv$ communicating processes performing computations
- Key feature: Blend of mathematical simulation and coarse-grain d. e. simulation
- Resources: Defined by a rate (MFlop/s or Mb/s) + latency
  - Also allows dynamic traces and failures
- Tasks can use multiple resources explicitely or implicitly
  - Transfer over multiple links, computation using disk and CPU
- Simple API to *specify* an heuristic or application easily

Casanova, Legrand, Quinson.
SimGrid: a Generic Framework for Large-Scale Distributed Experimentations, EUROSIM'08.

Experimental tools comparison

	CPU	Disk	Network	Application	Requirement	Settings	Scale
Grid'5000	direct	direct	direct	direct	access	fixed	<5000
Planetlab	virtualize	virtualize	virtualize	virtualize	none	uncontrolled	hundreds
Modelnet	-	-	emulation	emulation	lot material	controlled	dozens
MicroGrid	emulation	-	fine d.e.	emulation	none	controlled	hundreds
ns-2	-	-	fine d.e.	coarse d.e.	C++ and tcl	controlled	<1,000
SSFNet	-	-	fine d.e.	coarse d.e.	Java	controlled	<100,000
GTNetS	-	-	fine d.e.	coarse d.e.	C++	controlled	<177,000
ChicSim	coarse d.e.	-	coarse d.e.	coarse d.e.	C	controlled	few 1,000
OptorSim	coarse d.e.	amount	coarse d.e.	coarse d.e.	Java	controlled	few 1,000
GridSim	coarse d.e.	coarse d.e.	coarse d.e.	coarse d.e.	Java	controlled	few 1,000
P2PSim	-	-	-	state machine	C++	controlled	few 1,000
PlanetSim	-	-	cste time	coarse d.e.	Java	controlled	100,000
PeerSim	-	-	-	state machine	Java	controlled	1,000,000
SimGrid	math/d.e.	(underway)	math/d.e.	d.e./emul	C or Java	controlled	few 100,000

- Direct execution $\rightsquigarrow$ no experimental bias (?)
  Experimental settings fixed (between hardware upgrades), but not controllable
- Virtualization allows sandboxing, but no experimental settings control
- Emulation can have high overheads (but captures the overhead)
- Discrete event simulation is slow, but hopefully accurate
  To scale, you have to trade speed for accuracy

So what simulator should I use?

It really depends on your goal / resources

- Grid'5000 experiments very good . . . if have access and plenty of time
- PlanetLab does not enable reproducible experiments
- ModelNet, ns-2, SSFNet, GTNetS meant for networking experiments (no CPU)
- ModelNet requires some specific hardware setup
- MicroGrid simulations take a lot of time (although they can be parallelized)
- SimGrid's models have clear limitations (e.g. for short transfers)
- SimGrid simulations are quite easy to set up (but rewrite needed)
- SimGrid does not require that a full application be written
- Ad-hoc simulators are easy to setup, but their validity is still to be shown, *ie,* the results obtained *may* be plainly wrong
- Ad-hoc simulators obviously not generic (difficult to adapt to your own need)

Key trade-off seem to be accuracy *vs* speed

- The more abstract the simulation the fastest
- The less abstract the simulation the most accurate

Does this trade-off really hold?

Simulation Validation

Crux of simulation works

- Validation is difficult
- Almost never done convincingly
- (not specific to CS: other science have same issue here)

How to validate a model (and obtain scientific results?)

- Claim that it is plausible (justification = argumentation)
- Show that it is reasonable
  - Some validation graphs in a few special cases at best
  - Validation against another "validated" simulator
- Argue that trends are respected (absolute values may be off)
  $\leadsto$ it is useful to compare algorithms/designs
- Conduct extensive verification campaign against real-world settings

Simulation Validation: the FLASH example

FLASH project at Stanford

- Building large-scale shared-memory multiprocessors
- Went from conception, to design, to actual hardware (32-node)
- Used simulation heavily over 6 years

Authors compared simulation(s) to the real world

- Error is unavoidable (30% error in their case was not rare)
  Negating the impact of "we got 1.5% improvement"
- Complex simulators not ensuring better simulation results
  - Simple simulators worked better than sophisticated ones (which were unstable)
  - Simple simulators predicted trends as well as slower, sophisticated ones

  $\Rightarrow$ Should focus on simulating the important things
- Calibrating simulators on real-world settings is mandatory

For FLASH, the simple simulator was all that was needed...

Gibson, Kunz, Ofelt, Heinrich, *FLASH vs. (Simulated) FLASH: Closing the Simulation Loop*, Architectural Support for Programming Languages and Operating Systems, 2000

Conclusion

Large-Scale Distributed System Research is Experimental

- Analytical models are too limited
- Real-world experiments are hard & limited

$\Rightarrow$ Most literature rely on simulation

Simulation for distributed applications still taking baby steps

- Compared for example to hardware design or networking communities but more advanced for HPC Grids than for P2P
- Lot of home-made tools, no standard methodology
- Very few simulation projects even try to:
  - Publish their tools for others to use
  - Validate their tools
  - Support other people's use:
    genericity, stability, portability, documentation, ...

Conclusion

Claim: SimGrid may prove helpful to your research

- User-community much larger than contributors group
- Used in several communities (scheduling, GridRPC, HPC infrastructure, P2P)
- Model limits known thanks to validation studies
- Easy to use, extensible, fast to execute
- Around since almost 10 years

Remainder of this talk: present SimGrid in detail

- Under the cover:
  - Models used
- Implementation overview
  - SimGrid architecture
  - Features
- Main limitations
  - Tool performance and scalability
- Hands On
  - Scheduling algorithm experiences

Agenda

- Distributed Systems Experiments
  - Methodological Issues
  - Main Methodological Approaches
  - Tools for Experimentations in Large-Scale Distributed Systems
- Resource Models
  - Analytic Models Underlying SimGrid
- SimGrid Architecture and Features
  - Overview of the SimGrid Components
  - SimDag: Comparing Scheduling Heuristics for DAGs
  - MSG: Comparing Heuristics for Concurrent Sequential Processes
  - GRAS: Developing and Debugging Real Applications
  - SMPI: Running MPI applications on top of SimGrid
- Conclusion

Agenda

Analytic Models underlying the SimGrid Framework

Main challenges for SimGrid design

- Simulation accuracy:
  - Designed for HPC scheduling community $\rightsquigarrow$ don't mess with the makespan!
  - At the very least, understand validity range
- Simulation speed:
  - Users conduct large parameter-sweep experiments over alternatives

Microscopic simulator design

- Simulate the packet movements and routers algorithms
- Simulate the CPU actions (or micro-benchmark classical basic operations)
- Hopefully very accurate, but *very* slow (simulation time $\gg$ simulated time)

Going faster while remaining reasonable?

- Need to come up with macroscopic models for each kind of resource
- Main issue: resource sharing. *Emerge* naturally in microscopic approach:
  - Packets of different connections interleaved by routers
  - CPU cycles of different processes get slices of the CPU

Modeling a Single Resource

Basic model: $Time = L + \frac{size}{B}$

- Resource work at given *rate* (B, in MFlop/s or Mb/s)
- Each use have a given latency (L, in s)

Application to processing elements (CPU/cores)

- Very widely used (latency usually neglected)
- No cache effects and other specific software/hardware adequation
- No better analytical model (reality too complex and changing)
- Sharing easy in steady-state: fair share for each process

Application to networks

- Turns out to be "inaccurate" for TCP
- B not constant, but depends on RTT, packet loss ratio, window size, *etc.*
- Several models were proposed in the literature

Modeling TCP performance (single flow, single link)

Padhye, Firoiu, Towsley, Krusoe. *Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation.* IEEE/ACM Transactions on Networking, Vol. 8, Num. 2, 2000.

$$B = min\left(\frac{W_{max}}{RTT}, \frac{1}{RTT\sqrt{2bp/3} + T_0 \times min(1, 3\sqrt{3bp/8}) \times p(1+32p^2)}\right)$$

- W_{max}: receiver advertised window
- p: loss indication rate
- RTT: Round trip time
- b: #packages acknowledged per ACK
- T_0: TCP average retransmission timeout value

Model discussion

- Captures TCP congestion control (fast retransmit and timeout mecanisms)
- Assumes steady-state (no slow-start)
- Accuracy shown to be good over a wide range of values
- p and b not known in general (model hard to instanciable)

SimGrid model for single TCP flow, single link

Definition of the link l

- L_l: physical latency
- B_l: physical bandwidth

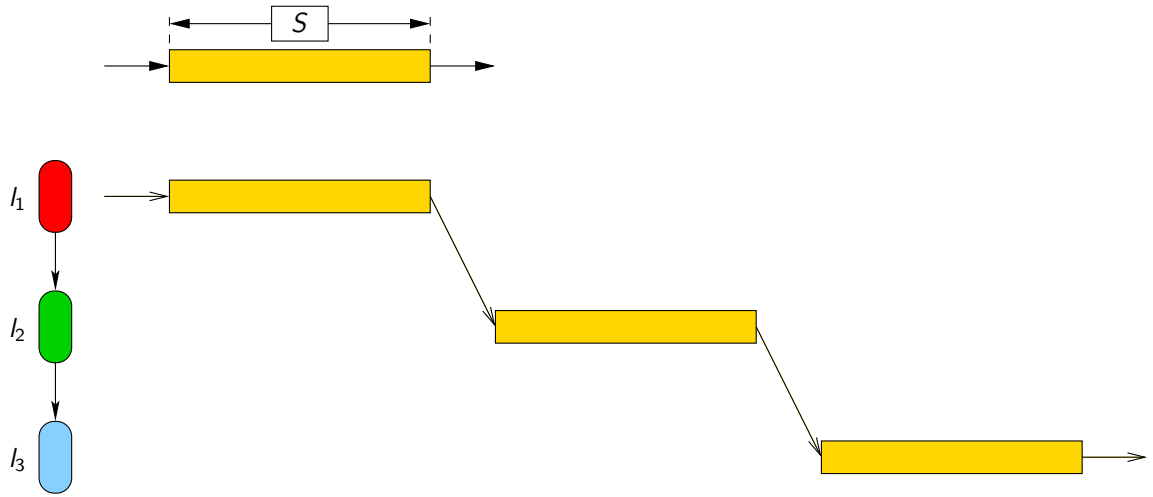
Time to transfer *size* bytes over the link:

$$Time = L_l + \frac{size}{B'_l}$$

Empirical bandwidth: $B'_l = min(B_l, \frac{W_{max}}{RTT})$

- Justification: sender emits W_{max} then waits for ack (ie, waits RTT)
- Upper limit: first *min* member of previous model
- RTT assumed to be twice the physical latency
- Router queue time assumed to be included in this value

Modeling Multi-hop Networks: Store & Forward



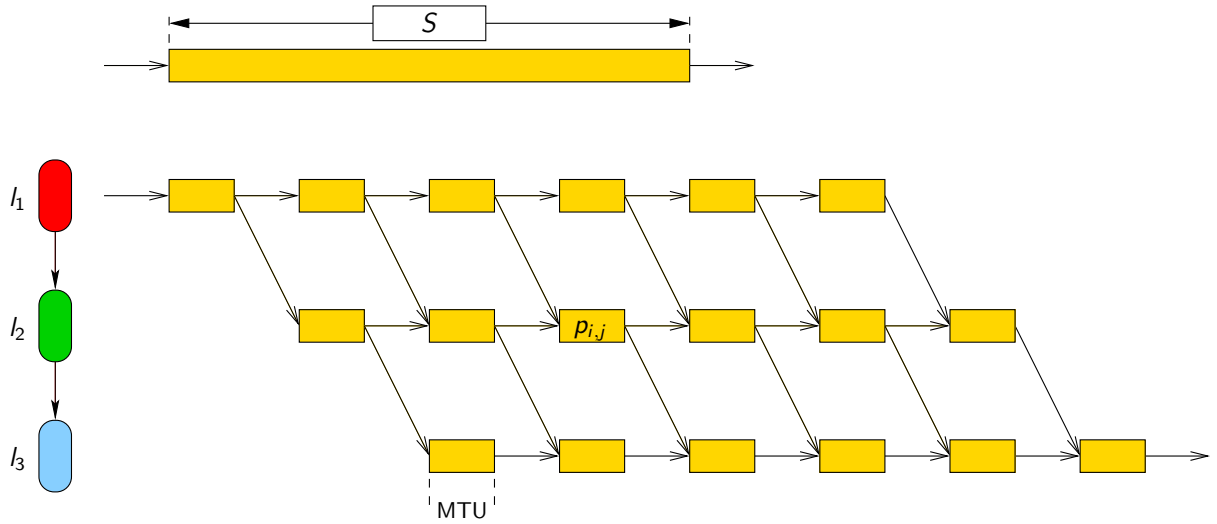
First idea, quite natural

- Pay the price of going through link 1, then go through link 2, etc.
- Analogy to the time to go from a city to another: time on each road

Unfortunately, things don't work this way

- Whole message not stored on each router
- Data split in packets over TCP networks (surprise, surprise)
- Transfers on each link occur in parallel

Modeling Multi-hop Networks: WormHole



Remember Networking classes?

- Links packetize stream according to MTU (Maximum Transmission Unit)
- Easy to simulate (SimGrid until 2002; GridSim 4.0 & most ad-hoc tools do)

Unfortunately, things don't work this way

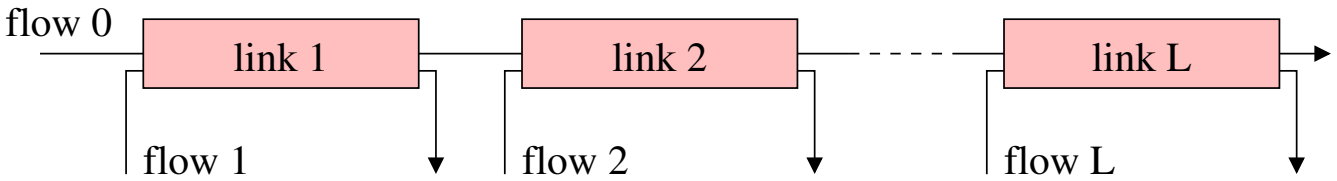
- IP packet fragmentation algorithms complex (when MTUs differ)
- TCP contention mecanisms:
  - Sender only emits *cwnd* packets before ACK
  - Timeouts, fast retransmit, etc.

$\Rightarrow$ as slow as packet-level simulators, not quite as accurate

Macroscopic TCP modeling is a field

TCP bandwidth sharing studied by several authors

- Data streams modeled as fluids in pipes
- Same model for single stream/multiple links or multiple stream/multiple links



Notations

- $\mathcal{L}$: set of links
- C_l: capacity of link l ($C_l > 0$)
- n_l: amount of flows using link l
- $\mathcal{F}$: set of flows; $f \in P(\mathcal{L})$
- λ_f : transfer rate of f

Feasibility constraint

- Links deliver their capacity at most: $\forall l \in \mathcal{L}, \sum_{f \ni l} \lambda_f \leq C_l$

Max-Min Fairness

Objective function: maximize $\min_{f \in \mathcal{F}}(\lambda_f)$

- Equilibrium reached if increasing any λ_f decreases a λ'_f (with $\lambda_f > \lambda'_f$)
- Very reasonable goal: gives fair share to anyone
- Optionally, one can add prorities w_i for each flow i
  $\rightsquigarrow$ maximizing $\min_{f \in \mathcal{F}}(w_f \lambda_f)$

Bottleneck links

- For each flow f, one of the links is the limiting one l
  (with more on that link l, the flow f would get more overall)
- The objective function gives that l is saturated, and f gets the biggest share

$$\forall f \in \mathcal{F}, \exists l \in f, \quad \sum_{f' \ni l} \lambda_{f'} = C_l \quad \text{and} \quad \lambda_f = \max\{\lambda_{f'}, f' \ni l\}$$

L. Massoulié and J. Roberts, *Bandwidth sharing: objectives and algorithms*, IEEE/ACM Trans. Netw., vol. 10, no. 3, pp. 320-328, 2002.

Implementation of Max-Min Fairness

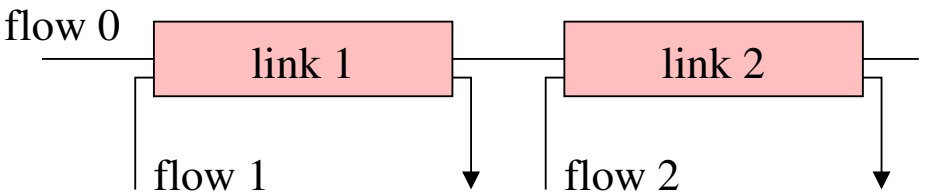
Bucket-filling algorithm

- Set the bandwidth of all flows to 0
- Increase the bandwidth of every flow by ϵ. And again, and again, and again.
- When one link is saturated, all flows using it are limited ($\rightsquigarrow$ removed from set)
- Loop until all flows have found a limiting link

Efficient Algorithm

1. Search for **the** bottleneck link l so that: $\frac{C_l}{n_l} = min\left\{\frac{C_k}{n_k},\ k \in \mathcal{L}\right\}$
2. $\forall f \in l,\ \ \lambda_f = \frac{C_l}{n_l}$;
   Update all n_l and C_l to remove these flows
3. Loop until all λ_f are fixed

Max-Min Fairness on Homogeneous Linear Network



$$C_1 = C \qquad n_1 = 2$$
$$C_2 = C \qquad n_2 = 2$$

$$\lambda_0 \leftarrow C/2$$
$$\lambda_1 \leftarrow C/2$$
$$\lambda_2 \leftarrow C/2$$

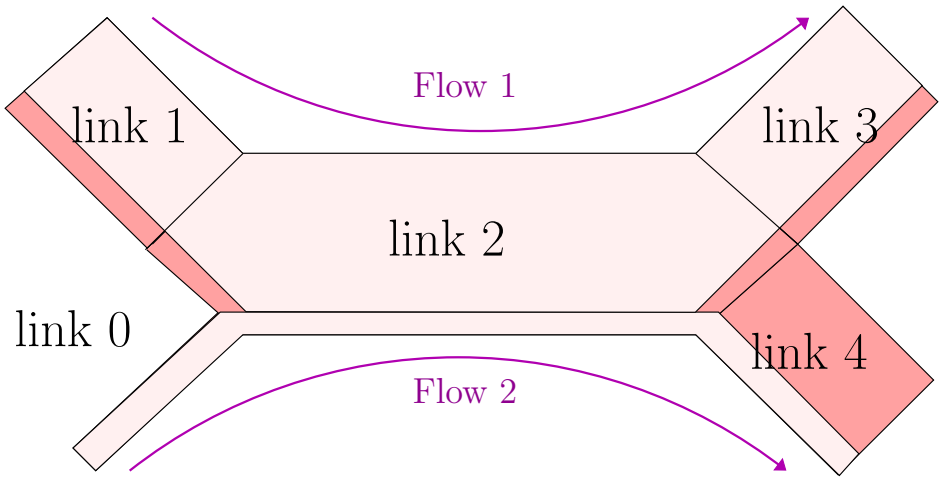
- All links have the same capacity C
- Each of them is limiting. Let's choose link 1

$\Rightarrow$ $\lambda_0 = C/2$ and $\lambda_1 = C/2$

- Remove flows 0 and 1; Update links' capacity
- Link 2 sets $\lambda_1 = C/2$

We're done computing the bandwidth allocated to each flow

Max-Min Fairness on Backbone



$$C_0 = 1 \qquad n_0 = 1$$
$$C_1 = 1000 \qquad n_1 = 1$$
$$C_2 = 1000 \qquad n_2 = 2$$
$$C_3 = 1000 \qquad n_3 = 1$$
$$C_4 = 1000 \qquad n_4 = 1$$

$$\lambda_1 \leftarrow 999$$
$$\lambda_2 \leftarrow 1$$

- The limiting link is link 0 $\left(\text{since } \frac{1}{1} = min\left(\frac{1}{1}, \frac{1000}{1}, \frac{1000}{2}, \frac{1000}{1}, \frac{1000}{1}\right)\right)$
- This fixes $\lambda_2 = 1$. Update the links
- The limiting link is link 2
- This fixes $\lambda_1 = 999$
- Done. We know λ_1 and λ_2

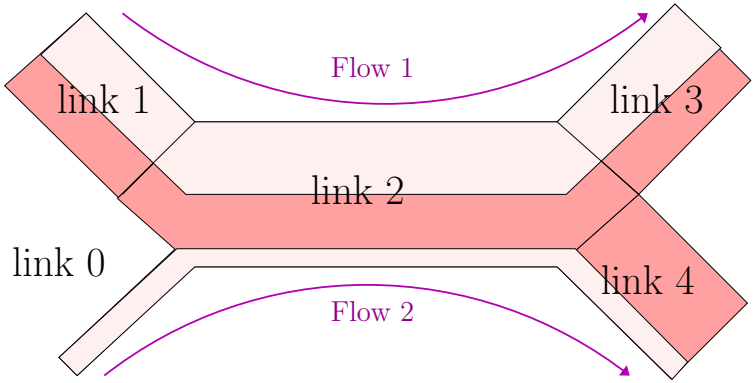
Side note: OptorSim 2.1 on Backbone

OptorSim (developped @CERN for Data-Grid)

- One of the rare ad-hoc simulators not using wormhole

Unfortunately, "strange" resource sharing:

1. For each link, compute the share that each flow may get: $\frac{C_l}{n_l}$
2. For each flow, compute what it gets: $\lambda_f = \min_{l \in f} \left(\frac{C_l}{n_l} \right)$



$$
\begin{array}{lll}
C_0 = 1 & n_1 = 1 & \text{share} = 1 \\
C_1 = 1000 & n_1 = 1 & \text{share} = 1000 \\
C_2 = 1000 & n_2 = 2 & \text{share} = 500 \\
C_3 = 1000 & n_3 = 1 & \text{share} = 1000 \\
C_4 = 1000 & n_4 = 1 & \text{share} = 1000
\end{array}
$$

$\lambda_1 = min(1000, 500, 1000) = \mathbf{500}$!!
$\lambda_2 = min(\ 1\ \ , 500, 1000) = 1$

λ_1 limited by link 2, but 499 still unused on link 2

This "unwanted feature" is even listed in the README file...

Proportional Fairness

Max-Min validity limits

- MaxMin gives a fair share to everyone
- Reasonable, but TCP does not do so
- Congestion mecanism: Additive Increase, Muplicative Decrease (AIMD)
- Complicates modeling, as shown in literature

Proportional Fairness

- MaxMin gives more to long flows (resource-eager), TCP known to do opposite
- Objective function: maximize $\sum_{\mathcal{F}} w_f \log(\lambda_f)$ (instead of $\min_{\mathcal{F}} w_f \lambda_f$ for MaxMin)
- *log* favors short flows

Kelly, *Charging and rate control for elastic traffic*, in European Transactions on Telecommunications, vol. 8, 1997, pp. 33-37.

Implementing Proportional Fairness

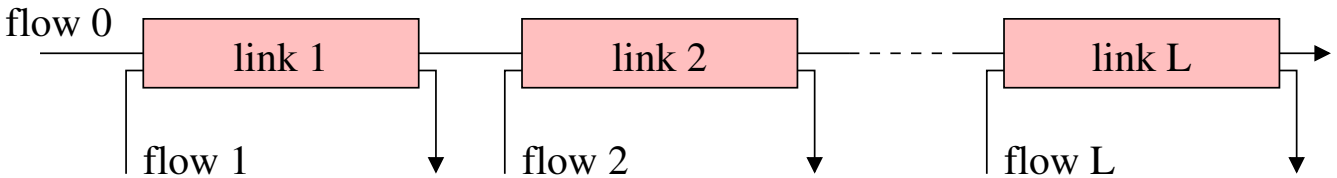
Karush Kuhn Tucker conditions:

- Solution $\{\lambda_f\}_{f\in\mathcal{F}}$ is uniq
- Any other feasible solution $\{\lambda'_f\}_{f\in\mathcal{F}}$ satisfy: $\sum_{f\in\mathcal{F}} \frac{\lambda'_f - \lambda_f}{\lambda_f} \leq 0$

$\Rightarrow$ Compute the point $\{\lambda_f\}$ where the derivate is zero (convex optimization)

$\rightarrow$ Use Lagrange multipliers and steepest gradient descent

Proportional Fairness on Homogeneous Linear Network



- Maths give that: $\lambda_0 = \frac{C}{n+1}$ and $\forall l \neq 0,\ \lambda_l = \frac{C \times n}{n+1}$
- Ie, for C=100Mb/s and n=3, $\lambda_0 = 25Mb/s$, $\lambda_1 = \lambda_2 = \lambda_3 = 75Mb/s$
- Closer to practitioner expectations

Recent TCP implementation

More protocol refinement, more model complexity

- Every agent changes its window size according to its neighbors' one (selfish net-utility maximization)
- Computing a distributed gradient for Lagrange multipliers $\rightsquigarrow$ same updates

TCP Vegas converges to a weighted proportional fairness

- Objective function: maximize $\sum L_f \times \log(\lambda_f)$ (L_f being the latency)

TCP Reno is even worse

- Objective function: maximize $\sum_{f \in \mathcal{F}} \arctan(\lambda_f)$

Low, S.H., *A Duality Model of TCP and Queue Management Algorithms*, IEEE/ACM Transactions on Networking, 2003.

Efficient implementation: possible, but not so trivial

- Computing *distributed* gradient for Lagrange multipliers: useless in our setting
- Lagrange multipliers computable with efficient optimal-step gradient descent

So, what is the model used in SimGrid?

"--cfg=network_model" command line argument

- `CM02` ⇝ MaxMin fairness
- `Vegas` ⇝ Vegas TCP fairness (Lagrange approach)
- `Reno` ⇝ Reno TCP fairness (Lagrange approach)
- By default in SimGrid v3.3: CM02
- Example: `./my_simulator --cfg=network_model:Vegas`

CPU sharing policy

- Default MaxMin is sufficient for most cases
- `cpu_model:ptask_L07` ⇝ model specific to parallel tasks

Want more?

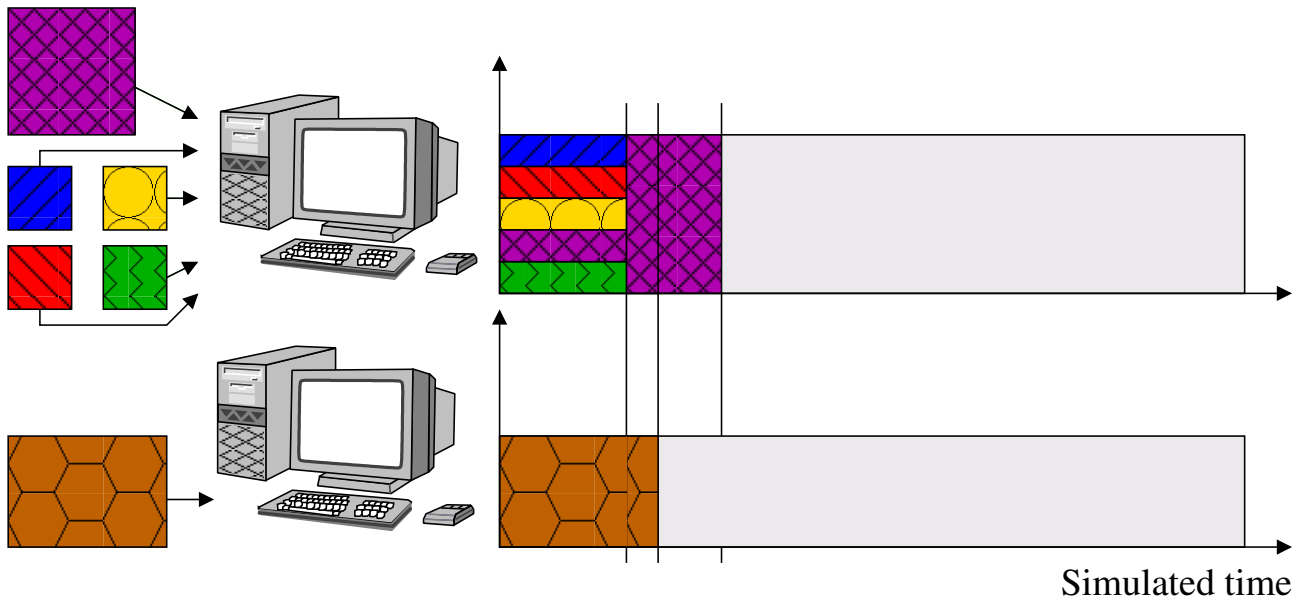
- `network_model:gtnets` ⇝ use Georgia Tech Network Simulator for network Accuracy of a packet-level network simulator without changing your code (!)
- Plug your own model in SimGrid!!
  (usable as scientific instrument in TCP modeling field, too)

How are these models used in practice?

Simulation kernel main loop

Data: set of resources with working rate

1. Some actions get created (by application) and assigned to resources
2. Compute share of everyone (resource sharing algorithms)
3. Compute the earliest finishing action, advance simulated time to that time
4. Remove finished actions
5. Loop back to 2



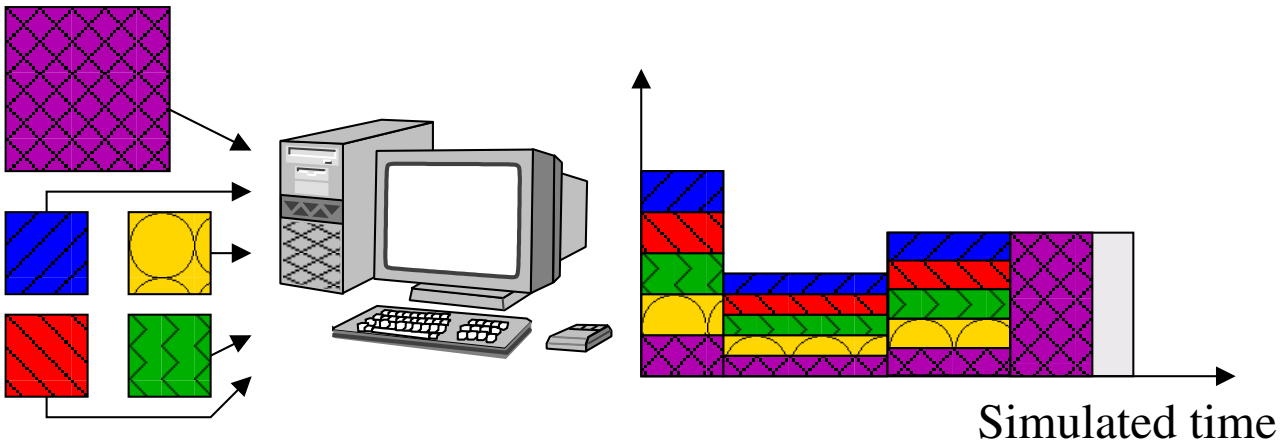
Adding Dynamic Availabilities to the Picture

Trace definition

- List of discrete events where the maximal availability changes
- $t_0 \rightarrow 100\%$, $t_1 \rightarrow 50\%$, $t_2 \rightarrow 80\%$, *etc.*

Adding traces doesn't change kernel main loop

- Availability changes: simulation events, just like action ends



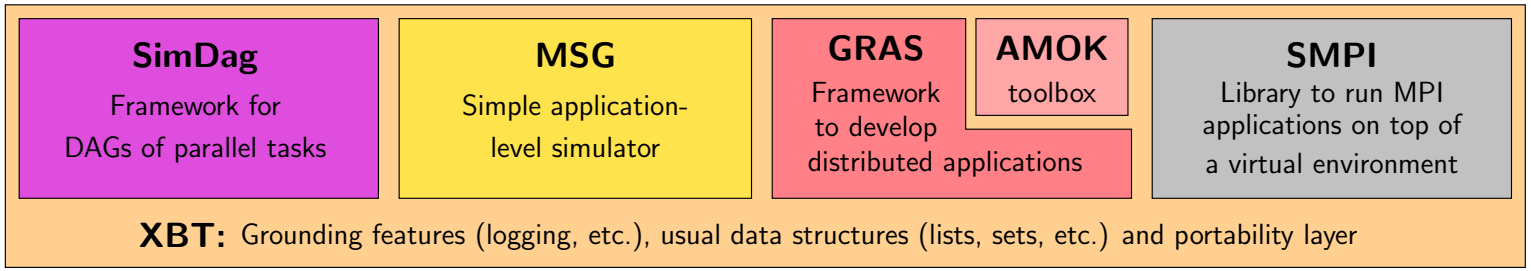
SimGrid also accept state changes (on/off)

Agenda

- Distributed Systems Experiments
  - Methodological Issues
  - Main Methodological Approaches
  - Tools for Experimentations in Large-Scale Distributed Systems
- Resource Models
  - Analytic Models Underlying SimGrid
- SimGrid Architecture and Features
  - Overview of the SimGrid Components
  - SimDag: Comparing Scheduling Heuristics for DAGs
  - MSG: Comparing Heuristics for Concurrent Sequential Processes
  - GRAS: Developing and Debugging Real Applications
  - SMPI: Running MPI applications on top of SimGrid
- Conclusion

Agenda

User-visible SimGrid Components



SimGrid user APIs

- SimDag: model applications as DAG of (parallel) tasks
- MSG: model applications as Concurrent Sequential Processes
- GRAS: develop real applications, studied and debugged in simulator
  AMOK: set of distributed tools (bandwidth measurement, *failure detector*, ...)
- SMPI: simulate MPI codes
- XBT: grounding toolbox

Which API should I choose?

- Your application is a DAG $\rightsquigarrow$ SimDag
- You have a MPI code $\rightsquigarrow$ SMPI
- You study concurrent processes, or distributed applications
  - You need performance graphs about several heuristics for a paper $\rightsquigarrow$ MSG
  - You develop a real application (or want experiments on real platform) $\rightsquigarrow$ GRAS
- Most popular API (for now): MSG

Argh! Do I really have to code in C?!

No, not necessary

- Some bindings exist: Java bindings to the MSG interface (new in v3.3)
- More bindings planned:
  - C++, Python, and any scripting language
  - SimDag interface

Well, sometimes yes, but...

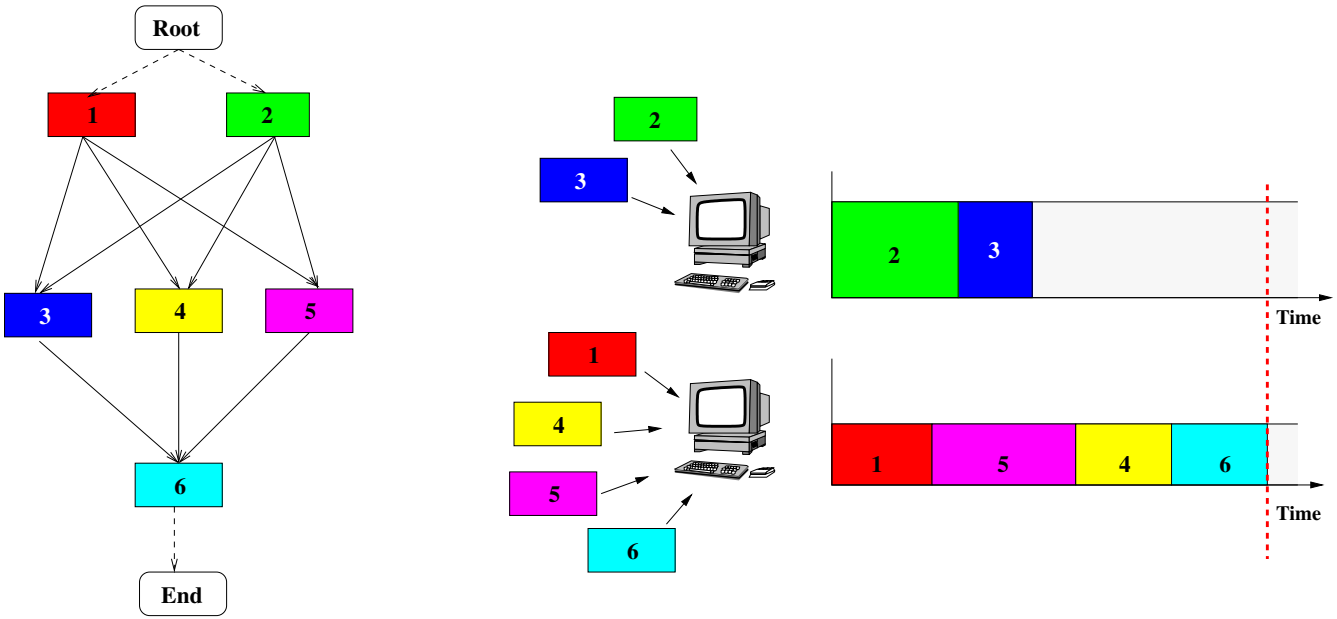
- SimGrid itself is written from C for speed and portability (no dependency)
- All components naturally usable from C (most of them only accessible from C)
- XBT eases some difficulties of C
  - Full-featured logs (similar to log4j), Exception support (in ANSI C)
  - Popular abstract data types (dynamic array, hash tables, ...)
  - Easy string manipulation, Configuration, Unit testing, ...

What about portability?

- Regularly tested under: Linux (x86, amd64), *Windows* and MacOSX
- Supposed to work under any other Unix system (including AIX and Solaris)

Agenda

SimDag: Comparing Scheduling Heuristics for DAGs



Main functionalities

1. Create a DAG of tasks
   - Vertices: tasks (either communication or computation)
   - Edges: precedence relation
2. Schedule tasks on resources
3. Run the simulation (respecting precedences)

   $\rightsquigarrow$ Compute the makespan

The SimDag interface

DAG creation

- Creating tasks: `SD_task_create(name, data)`
- Creating dependencies: `SD_task_dependency_{add/remove}(src,dst)`

Scheduling tasks

- `SD_task_schedule(task, workstation_number, *workstation_list, double *comp_amount, double *comm_amount, double rate)`
  - Tasks are parallel by default; simply put workstation_number to 1 if not
  - Communications are regular tasks, `comm_amount` is a matrix
  - Both computation and communication in same task possible
  - rate: To slow down non-CPU (resp. non-network) bound applications
- `SD_task_unschedule`, `SD_task_get_start_time`

Running the simulation

- `SD_simulate(double how_long)` (how_long $< 0 \rightsquigarrow$ until the end)
- `SD_task_{watch/unwatch}`: simulation stops as soon as task's state changes

Full API in the doxygen-generated documentation

Agenda

MSG: Heuristics for Concurrent Sequential Processes

(historical) Motivation

- Centralized scheduling does not scale
- SimDag (and its predecessor) not adapted to study decentralized heuristics
- MSG not strictly limited to scheduling, but particularly convenient for it

Main MSG abstractions

- **Agent:** some code, some private data, running on a given host
  set of functions + XML deployment file for arguments
- **Task:** amount of work to do and of data to exchange
  - `MSG_task_create(name, compute_duration, message_size, void *data)`
  - Communication: `MSG_task_{put,get}`, `MSG_task_Iprobe`
  - Execution: `MSG_task_execute`
    `MSG_process_sleep`, `MSG_process_{suspend,resume}`
- Host: location on which agents execute
- Mailbox: similar to MPI tags

The MSG master/workers example: the worker

The master has a large number of tasks to dispatch to its workers for execution

```
int worker(int argc, char *argv[ ]) {
  m_task_t task;                    int errcode;
  int id = atoi(argv[1]);
  char mailbox[80];

  sprintf(mailbox,"worker-%d",id);

  while(1) {
    errcode = MSG_task_receive(&task, mailbox);
    xbt_assert0(errcode == MSG_OK, "MSG_task_get failed");

    if (!strcmp(MSG_task_get_name(task),"finalize")) {
      MSG_task_destroy(task);
      break;
    }

    INFO1("Processing '%s'", MSG_task_get_name(task));
    MSG_task_execute(task);
    INFO1("'%s' done", MSG_task_get_name(task));
    MSG_task_destroy(task);
  }

  INFO0("I'm done. See you!");
  return 0;
}
```

The MSG master/workers example: the master

```
int master(int argc, char *argv[ ]) {
 int number_of_tasks = atoi(argv[1]);      double task_comp_size = atof(argv[2]);
 double task_comm_size = atof(argv[3]);    int workers_count = atoi(argv[4]);
 char mailbox[80];                         char buff[64];
 int i;

 /* Dispatching (dumb round-robin algorithm) */
 for (i = 0; i < number_of_tasks; i++) {
   sprintf(buff, "Task_%d", i);
   task = MSG_task_create(sprintf_buffer, task_comp_size, task_comm_size, NULL);
   sprintf(mailbox,"worker-%d",i % workers_count);
   INFO2("Sending ¨%s¨ to mailbox ¨%s¨", task->name, mailbox);
   MSG_task_send(task, mailbox);
 }

 /* Send finalization message to workers */
 INFO0("All tasks dispatched. Let's stop workers");
 for (i = 0; i < workers_count; i++)
  MSG_task_put(MSG_task_create("finalize", 0, 0, 0), workers[i], 12);

 INFO0("Goodbye now!"); return 0;
}
```

The MSG master/workers example: deployment file

Specifying which agent must be run on which host, and with which arguments

XML deployment file

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "surfxml.dtd">
<platform version="2">

  <!-- The master process (with some arguments) -->
  <process host="Tremblay" function="master">
     <argument value="6"/>         <!-- Number of tasks -->
     <argument value="50000000"/> <!-- Computation size of tasks -->
     <argument value="1000000"/>  <!-- Communication size of tasks -->
     <argument value="3"/>         <!-- Number of workers -->
  </process>

  <!-- The worker process (argument: mailbox number to use) -->
  <process host="Jupiter" function="worker"><argument value="0"/></process>
  <process host="Fafard"  function="worker"><argument value="1"/></process>
  <process host="Ginette" function="worker"><argument value="2"/></process>

</platform>
```

The MSG master/workers example: the `main()`

Putting things together

```
int main(int argc, char *argv[ ]) {

  /* Declare all existing agent, binding their name to their function */
  MSG_function_register("master", &master);
  MSG_function_register("worker", &worker);

  /* Load a platform instance */
  MSG_create_environment("my_platform.xml");
  /* Load a deployment file */
  MSG_launch_application("my_deployment.xml");

  /* Launch the simulation (until its end) */
  MSG_main();

  INFO1("Simulation took %g seconds",MSG_get_clock());
}
```

The MSG master/workers example: raw output

```
[Tremblay:master:(1) 0.000000] [example/INFO] Got 3 workers and 6 tasks to process
[Tremblay:master:(1) 0.000000] [example/INFO] Sending 'Task_0' to 'worker-0'
[Tremblay:master:(1) 0.147613] [example/INFO] Sending 'Task_1' to 'worker-1'
[Jupiter:worker:(2) 0.147613] [example/INFO] Processing 'Task_0'
[Tremblay:master:(1) 0.347192] [example/INFO] Sending 'Task_2' to 'worker-2'
[Fafard:worker:(3) 0.347192] [example/INFO] Processing 'Task_1'
[Tremblay:master:(1) 0.475692] [example/INFO] Sending 'Task_3' to 'worker-0'
[Ginette:worker:(4) 0.475692] [example/INFO] Processing 'Task_2'
[Jupiter:worker:(2) 0.802956] [example/INFO] 'Task_0' done
[Tremblay:master:(1) 0.950569] [example/INFO] Sending 'Task_4' to 'worker-1'
[Jupiter:worker:(2) 0.950569] [example/INFO] Processing 'Task_3'
[Fafard:worker:(3) 1.002534] [example/INFO] 'Task_1' done
[Tremblay:master:(1) 1.202113] [example/INFO] Sending 'Task_5' to 'worker-2'
[Fafard:worker:(3) 1.202113] [example/INFO] Processing 'Task_4'
[Ginette:worker:(4) 1.506790] [example/INFO] 'Task_2' done
[Jupiter:worker:(2) 1.605911] [example/INFO] 'Task_3' done
[Tremblay:master:(1) 1.635290] [example/INFO] All tasks dispatched. Let's stop workers.
[Ginette:worker:(4) 1.635290] [example/INFO] Processing 'Task_5'
[Jupiter:worker:(2) 1.636752] [example/INFO] I'm done. See you!
[Fafard:worker:(3) 1.857455] [example/INFO] 'Task_4' done
[Fafard:worker:(3) 1.859431] [example/INFO] I'm done. See you!
[Ginette:worker:(4) 2.666388] [example/INFO] 'Task_5' done
[Tremblay:master:(1) 2.667660] [example/INFO] Goodbye now!
[Ginette:worker:(4) 2.667660] [example/INFO] I'm done. See you!
[2.667660] [example/INFO] Simulation time 2.66766
```

The MSG master/workers example: colorized output

```
$ ./my_simulator | MSG_visualization/colorize.pl
[    0.000][  Tremblay:master    ] Got 3 workers and 6 tasks to process
[    0.000][  Tremblay:master    ] Sending 'Task_0' to 'worker-0'
[    0.148][  Tremblay:master    ] Sending 'Task_1' to 'worker-1'
[    0.148][   Jupiter:worker    ] Processing 'Task_0'
[    0.347][  Tremblay:master    ] Sending 'Task_2' to 'worker-2'
[    0.347][    Fafard:worker    ] Processing 'Task_1'
[    0.476][  Tremblay:master    ] Sending 'Task_3' to 'worker-0'
[    0.476][   Ginette:worker    ] Processing 'Task_2'
[    0.803][   Jupiter:worker    ] 'Task_0' done
[    0.951][  Tremblay:master    ] Sending 'Task_4' to 'worker-1'
[    0.951][   Jupiter:worker    ] Processing 'Task_3'
[    1.003][    Fafard:worker    ] 'Task_1' done
[    1.202][  Tremblay:master    ] Sending 'Task_5' to 'worker-2'
[    1.202][    Fafard:worker    ] Processing 'Task_4'
[    1.507][   Ginette:worker    ] 'Task_2' done
[    1.606][   Jupiter:worker    ] 'Task_3' done
[    1.635][  Tremblay:master    ] All tasks dispatched. Let's stop workers.
[    1.635][   Ginette:worker    ] Processing 'Task_5'
[    1.637][   Jupiter:worker    ] I'm done. See you!
[    1.857][    Fafard:worker    ] 'Task_4' done
[    1.859][    Fafard:worker    ] I'm done. See you!
[    2.666][   Ginette:worker    ] 'Task_5' done
[    2.668][  Tremblay:master    ] Goodbye now!
[    2.668][   Ginette:worker    ] I'm done. See you!
[    2.668][                     ] Simulation time 2.66766
```

Agenda

MSG bindings for Java: master/workers example

```
import simgrid.msg.*;
public class BasicTask extends simgrid.msg.Task {
    public BasicTask(String name, double computeDuration, double messageSize)
                                                              throws JniException {
        super(name, computeDuration, messageSize);
   }
}
public class FinalizeTask extends simgrid.msg.Task {
   public FinalizeTask() throws JniException {
      super("finalize",0,0);
  }
}
public class Worker extends simgrid.msg.Process {
   public void main(String[ ] args) throws JniException, NativeException {
      String id = args[0];

      while (true) {
         Task t = Task.receive("worker-" + id);
         if (t instanceof FinalizeTask)
            break;
         BasicTask task = (BasicTask)t;
         Msg.info("Processing '" + task.getName() + "'");
         task.execute();
         Msg.info("'" + task.getName() + "' done ");
       }
      Msg.info("Received Finalize. I'm done. See you!");
   }
}
```

MSG bindings for Java: master/workers example

```
import simgrid.msg.*;
public class Master extends simgrid.msg.Process {
   public void main(String[ ] args) throws JniException, NativeException {
      int numberOfTasks = Integer.valueOf(args[0]).intValue();
      double taskComputeSize = Double.valueOf(args[1]).doubleValue();
      double taskCommunicateSize = Double.valueOf(args[2]).doubleValue();
      int workerCount = Integer.valueOf(args[3]).intValue();

      Msg.info("Got "+  workerCount + " workers and " + numberOfTasks + " tasks.");

      for (int i = 0; i < numberOfTasks; i++) {
         BasicTask task = new BasicTask("Task_" + i ,taskComputeSize,taskCommunicateSize);
         task.send("worker-" + (i % workerCount));

         Msg.info("Send completed for the task " + task.getName() +
                  " on the mailbox 'worker-" + (i % workerCount) +  "'");
      }
      Msg.info("Goodbye now!");
   }
}
```

MSG bindings for Java: master/workers example

Rest of the story

- XML files (platform, deployment) not modified
- No need for a main() function glueing things together
  - Java introspection mecanism used for this
  - `simgrid.msg.Msg` contains an adapted `main()` function
  - Name of XML files must be passed as command-line argument
- Output very similar too

What about performance loss?

tasks \ workers		100	500	1,000	5,000	10,000
1,000	native	.16	.19	.21	.42	0.74
	java	.41	.59	.94	7.6	27.
10,000	native	.48	.52	.54	.83	1.1
	java	1.6	1.9	2.38	13.	40.
100,000	native	3.7	3.8	4.0	4.4	4.5
	java	14.	13.	15.	29.	77.
1,000,000	native	36.	37.	38.	41.	40.
	java	121.	130.	134.	163.	200.

- Small platforms: ok
- Larger ones: not quite...

Agenda

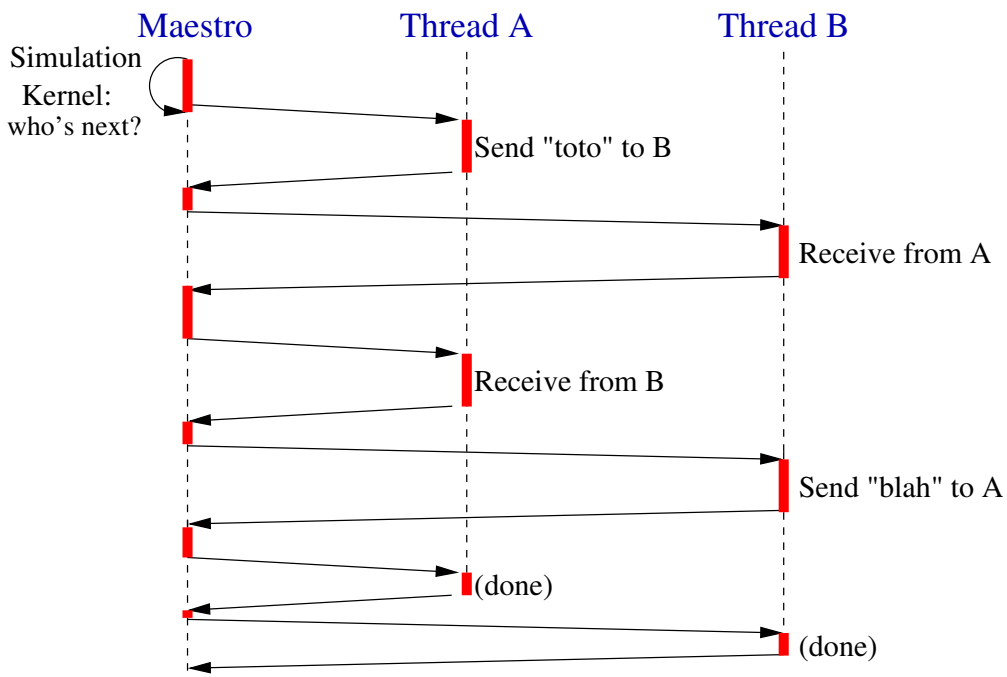
Implementation of CSPs on top of simulation kernel

Idea

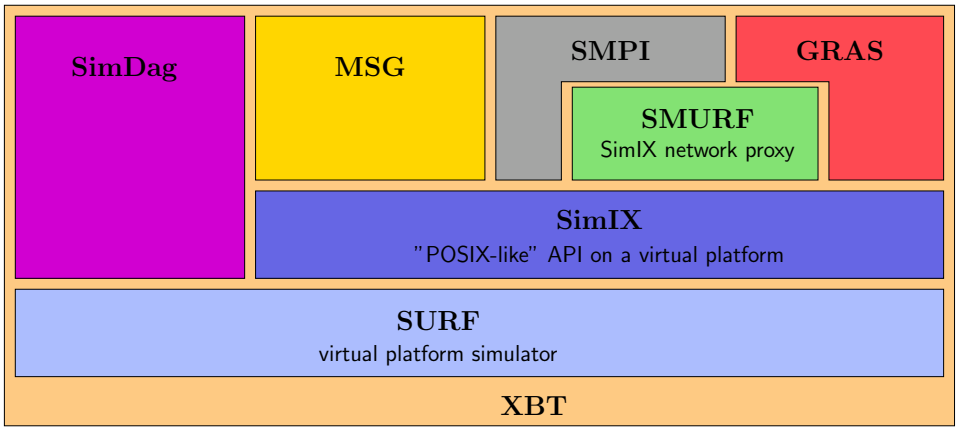
- Each process is implemented in a thread
- Blocking actions (execution and communication) reported into kernel
- A *maestro* thread unlocks the runnable threads (when action done)

Example

- Thread A:
  - Send "toto" to B
  - Receive something from B
- Thread B:
  - Receive something from A
  - Send "blah" to A
- Maestro schedules threads
  Order given by simulation kernel
- Mutually exclusive execution
  (don't fear)



A Glance at SimGrid Internals



- SURF: Simulation kernel, grounding simulation
  Contains all the models (uses GTNetS on need)
- SimIX: Eases the writting of user APIs based on CSPs
  Provided semantic: threads, mutexes and conditions on top of simulator
- SMURF: Allows to distribute the simulation over a cluster *(still to do)*
  Not for speed but for memory limit (at least for now)

▸ More on SimGrid internals

Agenda

Some Performance Results

Master/Workers on amd64 with 4Gb

#tasks	Context mecanism	#Workers					
		100	500	1,000	5,000	10,000	25,000
1,000	ucontext	0.16	0.19	0.21	0.42	0.74	1.66
	pthread	0.15	0.18	0.19	0.35	0.55	⋆
	java	0.41	0.59	0.94	7.6	27.	⋆
10,000	ucontext	0.48	0.52	0.54	0.83	1.1	1.97
	pthread	0.51	0.56	0.57	0.78	0.95	⋆
	java	1.6	1.9	2.38	13.	40.	⋆
100,000	ucontext	3.7	3.8	4.0	4.4	4.5	5.5
	pthread	4.7	4.4	4.6	5.0	5.23	⋆
	java	14.	13.	15.	29.	77.	⋆
1,000,000	ucontext	36.	37.	38.	41.	40.	41.
	pthread	42.	44.	46.	48.	47.	⋆
	java	121.	130.	134.	163.	200.	⋆

⋆: #semaphores reached system limit

(2 semaphores per user process,

System limit = 32k semaphores)

Extensibility with UNIX contextes

#tasks	Stack size	#Workers			
		25,000	50,000	100,000	200,000
1,000	128Kb	1.6	†	†	†
	12Kb	0.5	0.9	1.7	3.2
10,000	128Kb	2	†	†	†
	12Kb	0.8	1.2	2	3.5
100,000	128Kb	5.5	†	†	†
	12Kb	3.7	4.1	4.8	6.7
1,000,000	128Kb	41	†	†	†
	12Kb	33	33.6	33.7	35.5
5,000,000	128Kb	206	†	†	†
	12Kb	161	167	161	165

†: out of memory

Scalability limit of GridSim

- 1 user process = 3 java threads (code, input, output)
- System limit = 32k threads

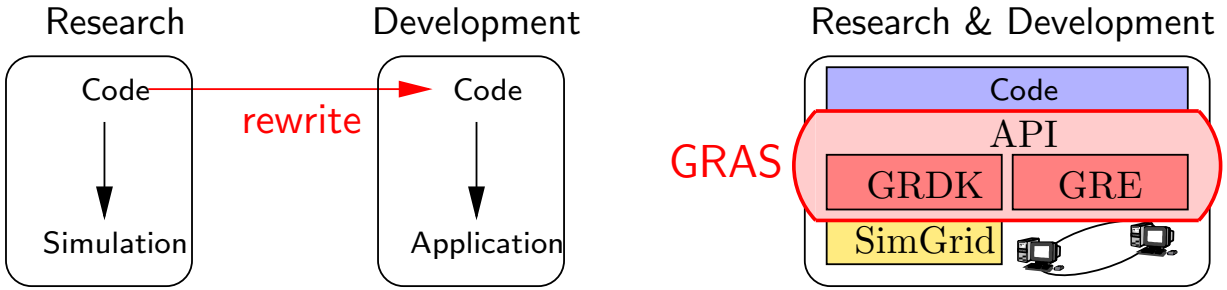
⇒ at most 10,922 user processes

Agenda

Goals of the GRAS project (Grid Reality And Simulation)

Ease development of large-scale distributed apps

Development of real distributed applications using a simulator



- Framework for Rapid Development of Distributed Infrastructure
  - Develop and tune on the simulator; Deploy *in situ* without modification
    How: One API, two implementations

- Efficient Grid Runtime Environment (result = application $\neq$ prototype)
  - Performance concern: efficient communication of structured data
    How: Efficient wire protocol (avoid data conversion)
  - Portability concern: because of grid heterogeneity
    How: ANSI C + autoconf + no dependency

Agenda

- Distributed Systems Experiments
  - Methodological Issues
  - Main Methodological Approaches
  - Tools for Experimentations in Large-Scale Distributed Systems
- Resource Models
  - Analytic Models Underlying SimGrid
- SimGrid Architecture and Features
  - Overview of the SimGrid Components
  - SimDag: Comparing Scheduling Heuristics for DAGs
  - MSG: Comparing Heuristics for Concurrent Sequential Processes
  - GRAS: Developing and Debugging Real Applications
    - Motivation and project goals
    - Functionalities
    - Experimental evaluation (performance and simplicity)
    - Conclusion and Perspectives
  - SMPI: Running MPI applications on top of SimGrid
- Conclusion

Main concepts of the GRAS API

Agents (acting entities)

- Code (C function)
- Private data
- Location (hosting computer)

Sockets (communication endpoints)

- Server socket: to receive messages
- Client socket: to contact a server (and receive answers)

Messages (what gets exchanged between agents)

- Semantic: Message type
- Payload described by data type description (fixed for a given type)

Callbacks (code to execute when a message is received)

- Also possible to explicitly wait for given messages

Emulation and Virtualization

Same code runs **without modification** both in simulation and *in situ*

- In simulation, agents run as threads within a single process
- In situ, each agent runs within its own process

$\Rightarrow$ Agents are threads, which can run as separate processes

Emulation issues

- How to get the process sleeping? How to get the current time?
  - System calls are *virtualized*: `gras_os_time`; `gras_os_sleep`
- How to report computation time into the simulator?
  - Asked explicitly by user, using provided macros
  - Time to report can be benchmarked automatically
- What about global data?
  - Agent status placed in a specific structure, ad-hoc manipulation API

Example of code: ping-pong (1/2)

Code common to client and server

```
#include "gras.h"
XBT_LOG_NEW_DEFAULT_CATEGORY(test,"Messages specific to this example");
static void register_messages(void) {
  gras_msgtype_declare("ping", gras_datadesc_by_name("int"));
  gras_msgtype_declare("pong", gras_datadesc_by_name("int"));
}
```

Client code

```
int client(int argc,char *argv[ ]) {
  gras_socket_t peer=NULL, from ;
  int ping=1234, pong;

  gras_init(&argc, argv);
  gras_os_sleep(1); /* Wait for the server startup */
  peer=gras_socket_client("127.0.0.1",4000);
  register_messages();

  gras_msg_send(peer, "ping", &ping);
  INFO3("PING(%d) -> %s:%d",ping, gras_socket_peer_name(peer), gras_socket_peer_port(peer));
  gras_msg_wait(6000,"pong",&from,&pong);

  gras_exit();
  return 0;
}
```

Example of code: ping-pong (2/2)

Server code

```
typedef struct { /* Global private data */
  int endcondition;
} server_data_t;
int server (int argc,char *argv[ ]) {
  server_data_t *globals;
  gras_init(&argc,argv);
  globals = gras_userdata_new(server_data_t);
  globals->endcondition=0;
  gras_socket_server(4000);
  register_messages();
  gras_cb_register("ping", &server_cb_ping_handler);

  while (!globals->endcondition) { /* Handle messages until our state change */
    gras_msg_handle(600.0);       /* Actually, one ping is enough for that */
  }
  free(globals);  gras_exit();  return 0;
}
int server_cb_ping_handler(gras_msg_cb_ctx_t ctx, void *payload_data) {
  server_data_t *globals = (server_data_t*)gras_userdata_get(); /* Get the globals */
  globals->endcondition = 1;

  int msg = *(int*) payload_data;                              /* What's the content? */
  gras_socket_t expeditor = gras_msg_cb_ctx_from(ctx);         /* Who sent it?*/
  /* Send data back as payload of a pong message to the ping's expeditor */
  gras_msg_send(expeditor, "pong", &msg);
  return 0;
}
```

Exchanging structured data

GRAS wire protocol: NDR (Native Data Representation)

Avoid data conversion when possible:

- Sender writes data on socket as they are in memory
- If receiver's architecture does match, no conversion
- Receiver able to convert from any architecture

GRAS message payload can be any valid C type

- Structure, enumeration, array, pointer, ...
- Classical garbage collection algorithm to deep-copy it
- Cycles in pointed structures detected & recreated

Describing a data type to GRAS

Manual description (excerpt)

```
gras_datadesc_type_t gras_datadesc_struct(name);
gras_datadesc_struct_append(struct_type,name,field_type);
gras_datadesc_struct_close(struct_type);
```

Automatic description of vector

```
GRAS_DEFINE_TYPE(s_vect,
 struct s_vect {
  int cnt;
  double*data GRAS_ANNOTE(size,cnt);
 }
);
```

C declaration stored into a `char*` variable to be parsed at runtime

Agenda

Assessing communication performance

Only communication performance studied since computation are not mediated

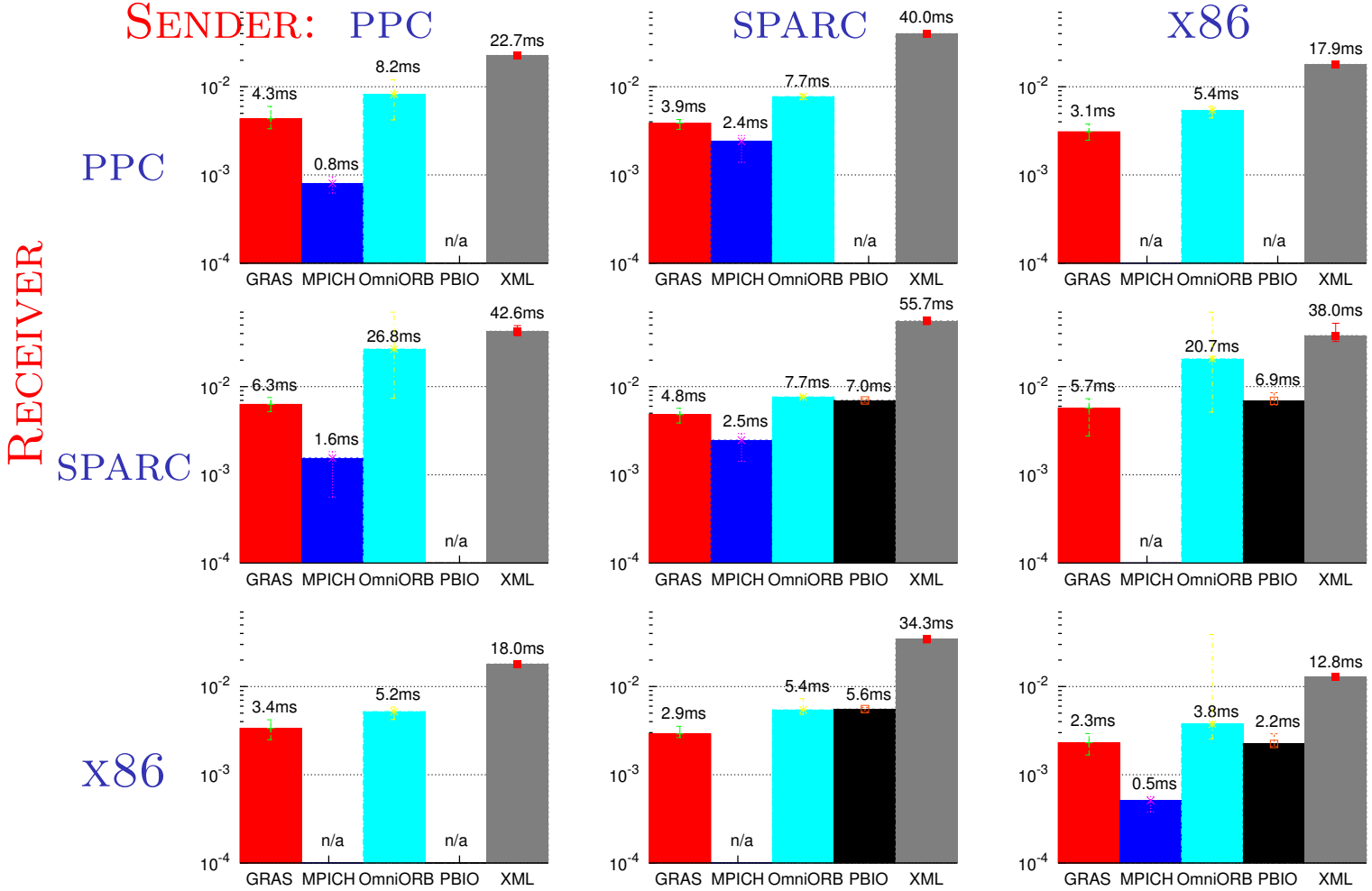
- Experiment: timing ping-pong of structured data (a message of Pastry)

```
typedef struct {
  int id, row_count;
  double time_sent;
  row_t *rows;
  int leaves[MAX_LEAFSET];
} welcome_msg_t;
```

```
typedef struct {
  int which_row;
  int row[COLS][MAX_ROUTESET];
} row_t;
```

- Tested solutions
  - GRAS
  - PBIO (uses NDR)
  - OmniORB (classical CORBA solution)
  - MPICH (classical MPI solution)
  - XML (Expat parser + handcrafted communication)
- Platform: x86, PPC, sparc (all under Linux)

Performance on a LAN



- MPICH twice as fast as GRAS, but cannot mix little- and big-endian Linux
- PBIO broken on PPC
- XML much slower (extra conversions + verbose wire encoding)

GRAS is the better compromise between performance and portability

Assessing API simplicity

Experiment: ran code complexity measurements on code for previous experiment

	GRAS	MPICH	PBIO	OmniORB	XML
McCabe Cyclomatic Complexity	8	10	10	12	35
Number of lines of code	48	65	84	92	150

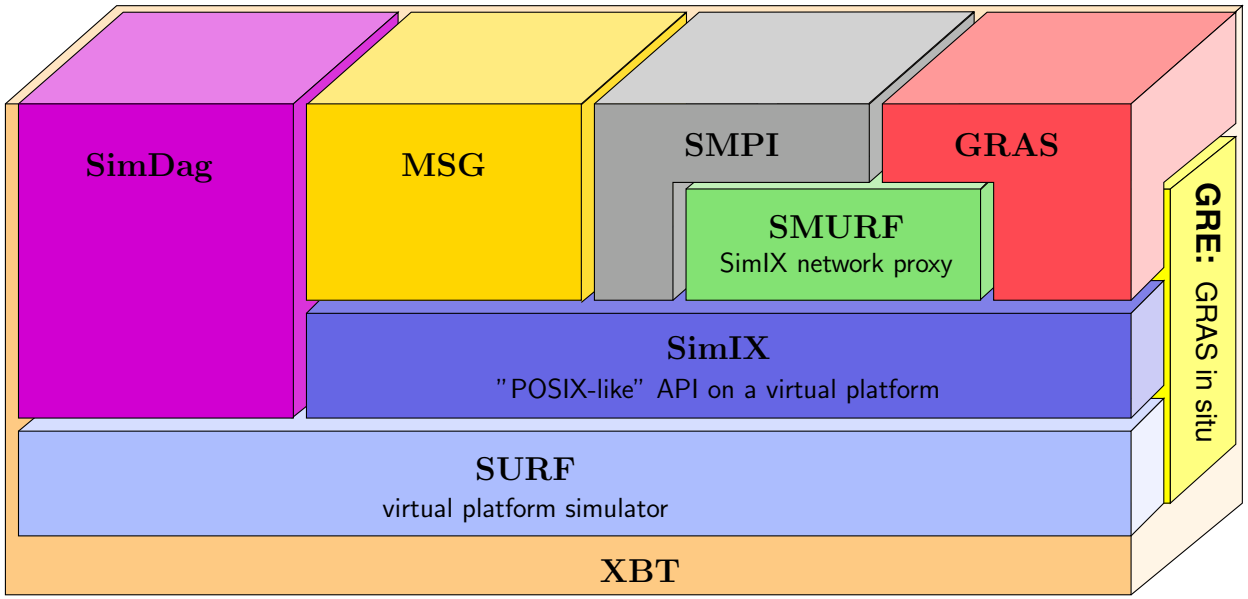
Results discussion

- XML complexity may be artefact of Expat parser (but fastest)
- MPICH: manual marshaling/unmarshalling
- PBIO: automatic marshaling, but manual type description
- OmniORB: automatic marshaling, IDL as type description
- GRAS: automatic marshaling & type description (IDL is C)

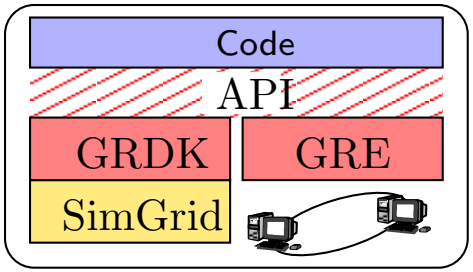
Conclusion

GRAS is the least demanding solution from developer perspective

Conclusion: GRAS eases infrastructure development



Research & Development



With GRAS

GRDK: Grid Research & Development Kit

- API for (explicitly) distributed applications
- Study applications in the comfort of the simulator

GRE: Grid Runtime Environment

- Efficient: twice as slow as MPICH, faster than OmniORB, PBIO, XML
- Portable: Linux (11 CPU archs); *Windows*; Mac OS X; Solaris; IRIX; AIX
- Simple and convenient:
  - API simpler than classical communication libraries (+XBT tools)
  - Easy to deploy: C ANSI; no dependency; autotools; <400kb

GRAS perspectives

Future work on GRAS

- Performance: type precompilation, communication taming and compression
- GRASPE (GRAS Platform Expender) for automatic deployment
- Model-checking as third mode along with simulation and in-situ execution

Details

Ongoing applications

- Comparison of P2P protocols (Pastry, Chord, etc)
- Use emulation mode to validate SimGrid models
- Network mapper (ALNeM): capture platform descriptions for simulator
- Large scale mutual exclusion service

Future applications

- Platform monitoring tool (bandwidth and latency)
- Group communications & RPC; Application-level routing; *etc.*

Agenda

- Distributed Systems Experiments
  - Methodological Issues
  - Main Methodological Approaches
  - Tools for Experimentations in Large-Scale Distributed Systems
- Resource Models
  - Analytic Models Underlying SimGrid
- SimGrid Architecture and Features
  - Overview of the SimGrid Components
  - SimDag: Comparing Scheduling Heuristics for DAGs
  - MSG: Comparing Heuristics for Concurrent Sequential Processes
  - GRAS: Developing and Debugging Real Applications
  - SMPI: Running MPI applications on top of SimGrid
- Conclusion

SMPI: Running MPI applications on top of SimGrid

Motivations

- Reproducible experimentation of MPI code (debugging)
- Test MPI code on still-to-build platform (dimensioning)

How it works

- `smpicc` changes MPI calls into SMPI ones (gettimeofday also intercepted)
- `smpirun` starts a classical simulation obeying `-hostfile` and `-np`

$\Rightarrow$ Runs unmodified MPI code after recompilation

Implemented calls

- Isend; Irecv. Recv; Send. Wait; Waitall; Waitany.
- Barrier; Bcast; Reduce; Allreduce (cmd line option to choose binary or flat tree)
- Comm_size; Comm_rank; Comm_split. Wtime. Init; Finalize; Abort.

Future Work

- Implement the rest of the API
- Test it more througfully
- Use it to validate SimGrid at application level (with NAS *et Al.*)

Agenda

- Distributed Systems Experiments
  - Methodological Issues
  - Main Methodological Approaches
  - Tools for Experimentations in Large-Scale Distributed Systems
- Resource Models
  - Analytic Models Underlying SimGrid
- SimGrid Architecture and Features
  - Overview of the SimGrid Components
  - SimDag: Comparing Scheduling Heuristics for DAGs
  - MSG: Comparing Heuristics for Concurrent Sequential Processes
  - GRAS: Developing and Debugging Real Applications
  - SMPI: Running MPI applications on top of SimGrid
- Conclusion

Conclusions on Distributed Systems Research

Research on Large-Scale Distributed Systems

- Reflexion about common methodologies needed (reproductible results needed)
- Purely theoritical works limited (simplistic settings $\leadsto$ NP-complete problems)
- Real-world experiments time and labor consuming; limited representativity
- Simulation appealing, if results remain validated

Simulating Large-Scale Distributed Systems

- Packet-level simulators too slow for large scale studies
- Large amount of ad-hoc simulators, but discutable validity
- Coarse-grain modelization of TCP flows possible (cf. networking community)
- Model instantiation (platform mapping or generation) remains challenging

SimGrid provides interesting models

- Implements non-trivial coarse-grain models for resources and sharing
- Validity results encouraging with regard to packet-level simulators
- Several orders of magnitude faster than packet-level simulators
- Several models availables, ability to plug new ones or use packet-level sim.

SimGrid provides several user interfaces

SimDag: Comparing Scheduling Heuristics for DAGs of (parallel) tasks

- Declare tasks, their precedences, schedule them on resource, get the makespan

MSG: Comparing Heuristics for Concurrent Sequential Processes

- Declare independent agents running a given function on an host
- Let them exchange and execute tasks
- Easy interface, rapid prototyping, Java bindings
- New in SimGrid v3.3.1: Trace-driven simulations

GRAS: Developing and Debugging Real Applications

- *Develop once, run in simulation or in situ* (debug; test on non-existing platforms)
- Resulting application twice slower than MPICH, faster than omniorb
- Highly portable and easy to deploy

SMPI: Running MPI applications on top of SimGrid (new in 3.3.1)

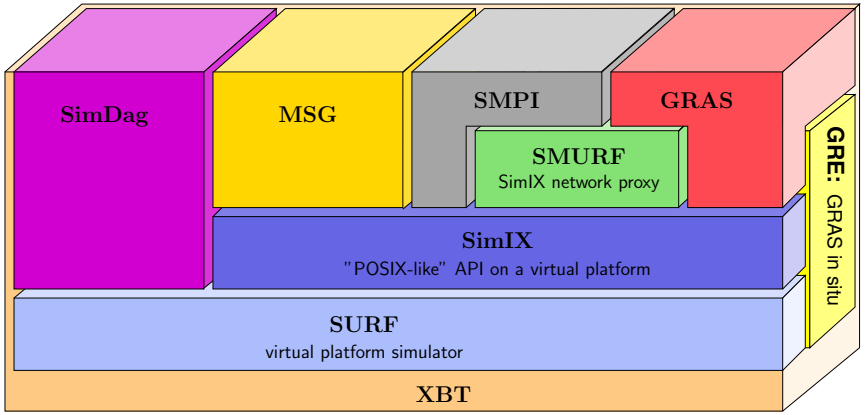
- Runs unmodified MPI code after recompilation (still partial implementation)

Other interfaces possible: OpenMP, BSP-like (any volunteer?)

SimGrid is an active and exciting project

Future Plans

- Improve usability
  (statistics tools, campain management)
- Extreme Scalability for P2P
- Model-checking of GRAS applications
- Emulation solution *à la* MicroGrid



Large community

http://gforge.inria.fr/projects/simgrid/

- 130 subscribers to the user mailling list (40 to -devel)
- 40 scientific publications using the tool for their experiments
  - 15 co-signed by one of the core-team members
  - 25 purely external
- LGPL, 120,000 lines of code (half for examples and regression tests)
- Examples, documentation and tutorials on the web page

Use it in your works!

Detailed agenda

Appendix (extra material)

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Model-checking GRAS application (ongoing work)

Executive Summary

Motivation

- GRAS allows to debug an application on simulator and deploy it when it works
- Problem: when to decide that *it works?*
  - Demonstrate a theorem → conversion to C difficult
  - Test some cases → may still fail on other cases

Model-checking

- Given an initial situation ("we have three nodes"),
  test all possible executions ("A gets first message first", "B does", "C does", ...)
- Combinatorial search in the tree of possibilities
- Fight combinatorial explosion: cycle detection, symmetry, abstraction

Model-checking in GRAS

- First difficulty: Checkpoint simulated processes (to rewind simulation)
  Induced difficulty: Devise **when** to checkpoint processes
- Second difficulty: Fight against combinatorial explosion

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Formal methods

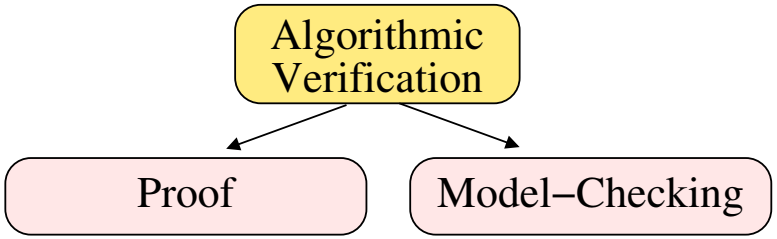
Goal: Develop safe software using automated methods

- Strong mathematical background
- Safe $\equiv$ respect some given properties

Kind of properties shown

- Safety: the car does not start without the key
- Liveness: if I push the break paddle, the car will eventually stop

Existing Formal Methods



Proof of programs

- In theory, applicable to any class of program
- In practice, quite tedious to use
  often limited to help a specialist doing the actual work (system state explosion)

Model-checking

- Shows that a system:

(safety) never evolves to a faulty state from a given initial state
(liveness) always evolve to the wanted state (stopping) from a given state (breaking)

- ☹ Less generic than proof: lack of faulty states for all initial state?
- ☺ Usable by non-specialists (at least, by *less-specialists*)

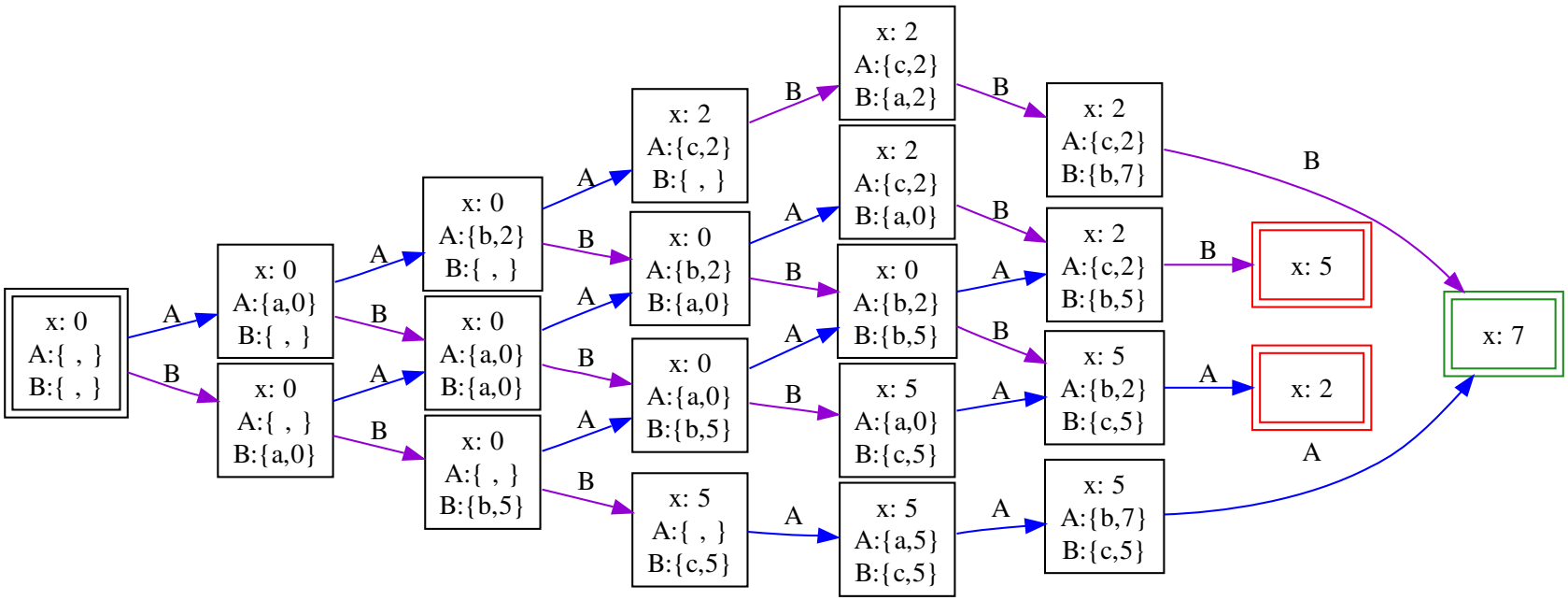
Example of problem to detect: Race Condition

x is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result : $x = 7$

a. Read the value of shared variable x and store it locally
b. Modify the local value (add 5 or 2)
c. Propagate the local variable into the shared one

- Execution of *Alice* **then** *Bob* or opposite: result = 7
- Interleaved execution: result = 2 or 5 (depending on last propagator)

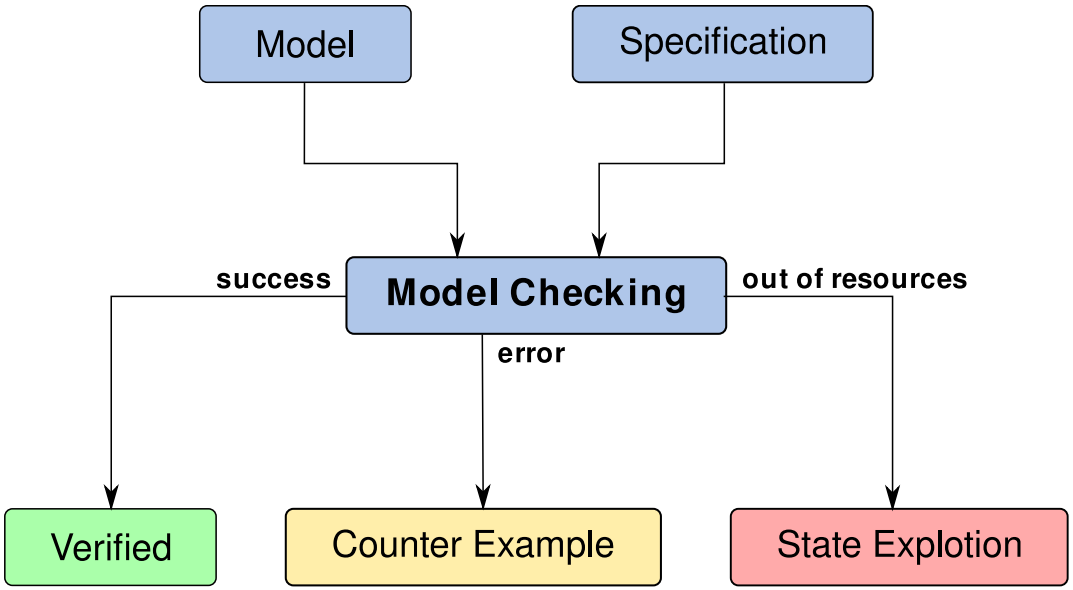
Model-checking: traverse graph of executions checking for properties



- Safety: assertions on each node
- Liveness by studying graph (cycle?)

Model-Checking Big Picture

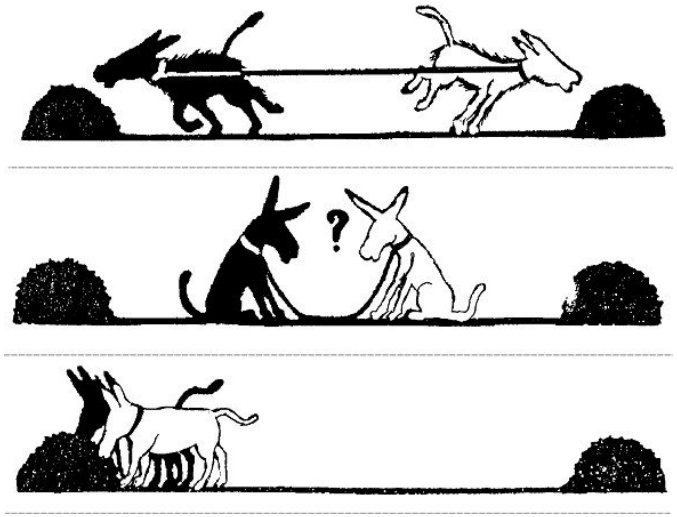
1. User writes Model (formal writing of algorithm) and Specification (set of properties)
2. Each decision point in model (if, input data) $\leadsto$ a branch in model state space
3. Check safety properties on each encountered node (state)
4. Store encountered nodes (to avoid looping) and transitions (to check liveness)
5. Process until:
   - State space completely traversed ($\Rightarrow$ model verified against this specification)
   - One of the property does not hold (the path until here is a counter-example)
   - We run out of resource ("state space explosion")



Classical field of application

Concurrent systems (ie, multithreaded; shared memory)

- Race condition: the result depends on execution order
- Deadlock: infinite wait between processes
- Starvation: one process prevented from using a free resource



Project goal: Extend to distributed systems

- Very little work done in this area
- Collaboration with Mosel team of INRIA

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Adding Model-Checking to SimGrid

Difficulties in Distributed System

- Race condition, Deadlock and Starvation, just as in concurrent algorithms
- Lack of global state: only local information available
- Asynchronism: no bound on communication time $\leadsto$ hard to detect failures

$\Rightarrow$ Model-checker for distributed algorithms appealing

But wait a minute...

Wasn't the simulator meant to test distributed algorithm already?!

- Simulation is better than real deployment because it is deterministic
- But possibly very low code coverage
- Model-Checking improves this, and provides counter-examples

$\leadsto$ Simulation to assess performance, Model-checking to assess correctness

Do not merge 2 tools in 1 and KISS instead!

- Avoid manual translation between formalisms to avoid introduction of errors
- Simulator and model-checker both need to:
  - Simulate of the environment (processes, network, messages)
  - Control over the scheduling of the processes
  - Intercept the communication

Identifying state transitions in SimGrid

Main MC challenge

- Concurrent systems lead to many interleavings
- Rapidly lead to state space explosion
  (don't even dream of model-checking a system with as much as 10 processes)

Good news in our context:

- Processes' state spaces are isolated; interaction only through message passing
  partial history, think of Lamport's clocks

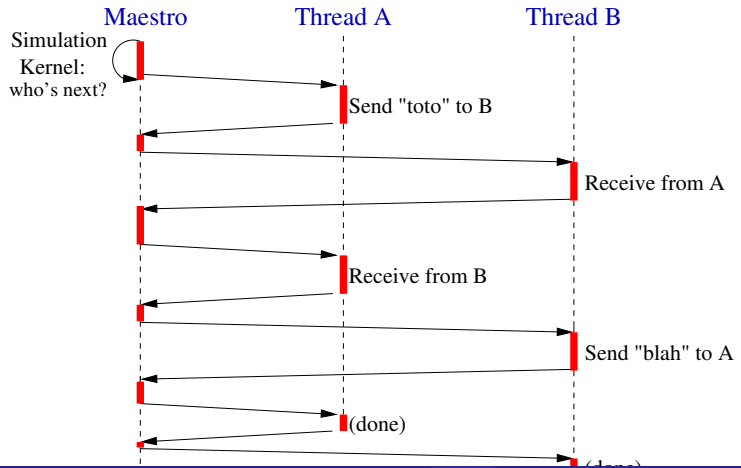
$\Rightarrow$ No need to consider all possible instruction interleavings
$\rightsquigarrow$ massive state space reduction is possible (but open research question)

Considered transitions in SimGrid

- Messages' send/receive only
- Coincide with simulation scheduling points!

▸ See it again

- Model-Checking logic can go in the maestro



Traversing the State Graph

```
for (i=0; i<slave_count; i++)
  if (power[i] > 12)
    send(big_task);
  else
    send(small_task);
```

How to follow several execution paths?

- Need to save & restore the state in order to rewind the application
- Most existing tool work by somehow interpreting a DSL
- Other are state-less (rerun the app from the start)
- In C, we need system support

Checkpointing threads

- Intercept memory allocation functions; use a special dynamic memory manager
  `#define malloc(a) my_malloc(a) /* and friends */`
  Deroute heap in separate memory segments (with shm ops)
- Also save/restore the stack (memcpy) and registers (setjmp) of the processes

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Current Status and Future Work

We implemented a first prototype

- Works for simple unmodified C GRAS programs
- Checks **safety** properties (C boolean functions; limited by C scoping rules)
- Simple state saving/restoring mechanism (process-wide)
- Simple depth-first search exploration (no search heuristic yet)

Current Work

- Isolating each simulated process's address space
- Separating the network in a separated address space
- Support the other SimGrid APIs

Future Work

- Exploit the heap symmetries (heap canonicalization)
- Implement partial order reductions
- Verification of LTL properties

Agenda

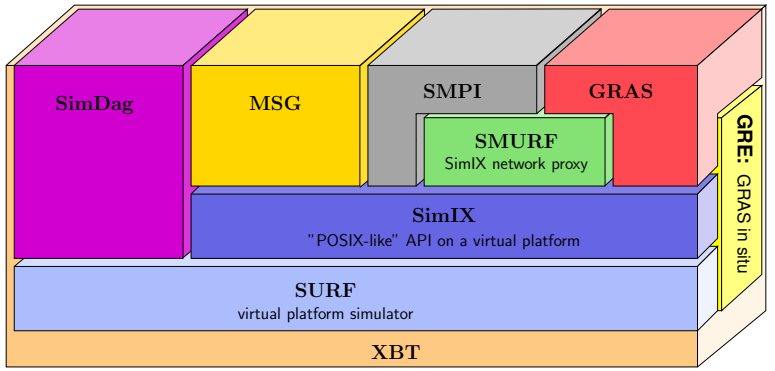
- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work

- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

SimGrid Internals

Some Numbers

- v3.3 is 120k sloc (w/o blanks; with comments)
  (Core lib: 47k; Tests: 69k; Doc:3.5k; +Build)
- v3.2 was 55k sloc
  (Core lib: 30k; Tests: 21k; Doc:3k; +Build)



SimGrid is quite strictly layered (and built bottom-up)

- XBT: Extensive toolbox
  Logs, exceptions, backtraces, ADT, strings, configuration, testing; Portability
- SURF: Simulation kernel, grounding simulation
  Main concepts: Resources (providing power) and actions (using resources)
- SimIX: Eases the writing of user APIs based on CSPs
  Adds concepts: Processes (running user code), plus mutex and conditions for synchro
- User interfaces: adds API-specific concepts
  (gras data description or MSG mailboxes; SimDag different: directly on top of SURF)

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
    - Big Picture
    - Models
    - How Models get used
    - Actions and Resources
    - Writing your own model
    - Adding new kind of models
  - Simix
  - Global Elements
  - Simix Process

SURF

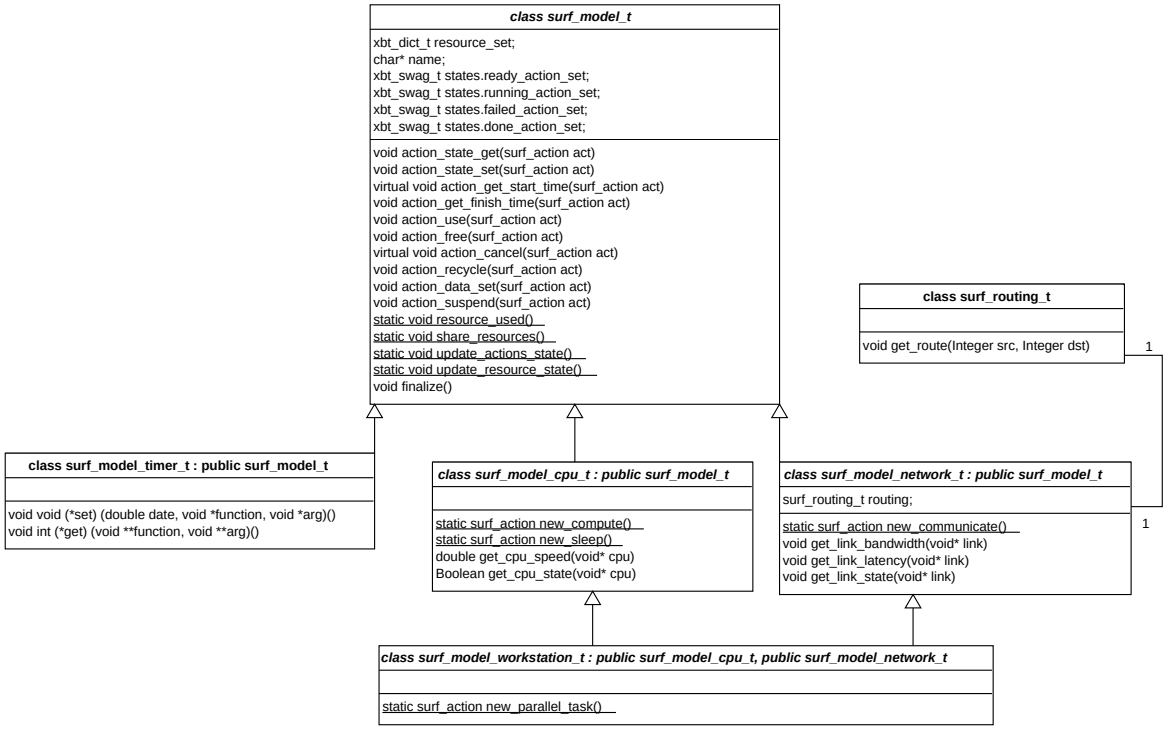
Big picture

- Resources provide power; created through the XML platform file
- Actions are consumption of this power; created by upper layers
- Designed with model extensibility and simulation efficiency in mind
  $\Rightarrow$ object oriented, but not in a canonical way

Models

- They express how resources get consumed by actions
- Act as class for action objects (heavy use of function pointers), and as fabric
- Several kind of models exist (one of each kind is in use during simulation)
  - CPU model: fabric of `compute` and `sleep` actions
  - Link model: fabric of `communicate` actions
  - Workstation model: aggregation pattern of CPU and link
  - Routing: not exactly a model; provide the list of links between two hosts

Models class diagram



Existing models (implement the abstract methods)

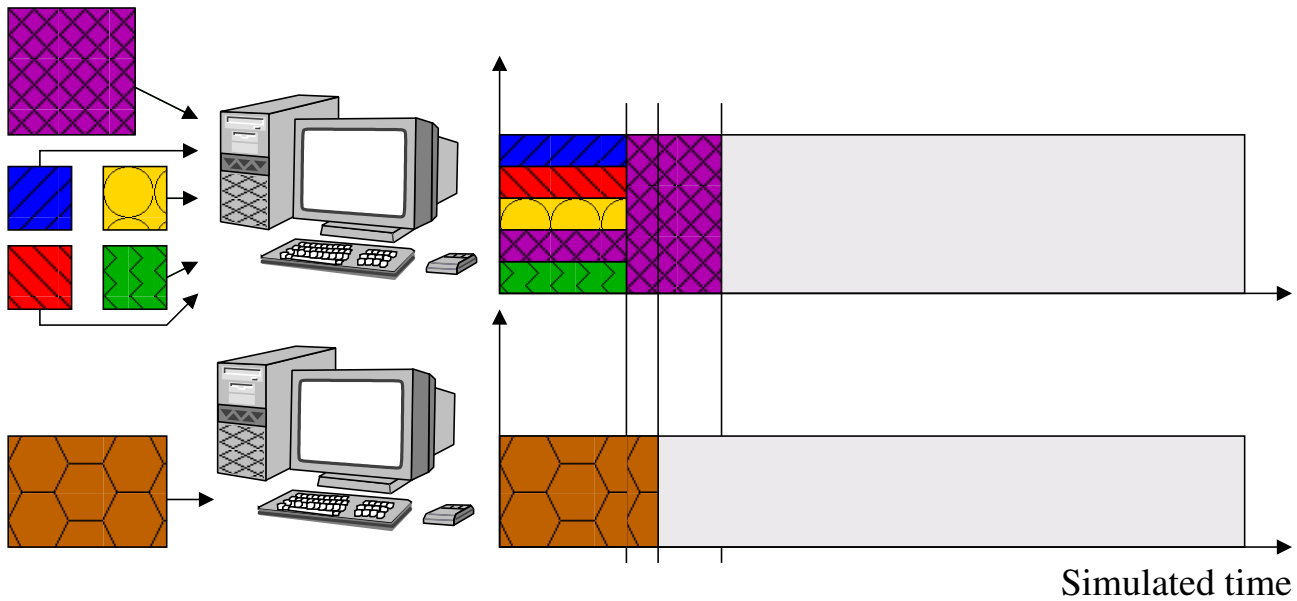
- Timer: only basic one
- CPU: Cas01 (regular maxmin)
- Link: CM02, LV08, Reno, Vegas; Constant, GTNetS
- Workstation: workstation, L07

How are these models used in practice?

Simulation kernel main loop

Data: set of resources with working rate

1. Some actions get created (by application) and assigned to resources
2. Compute share of everyone (resource sharing algorithms)
3. Compute the earliest finishing action, advance simulated time to that time
4. Remove finished actions
5. Loop back to 2



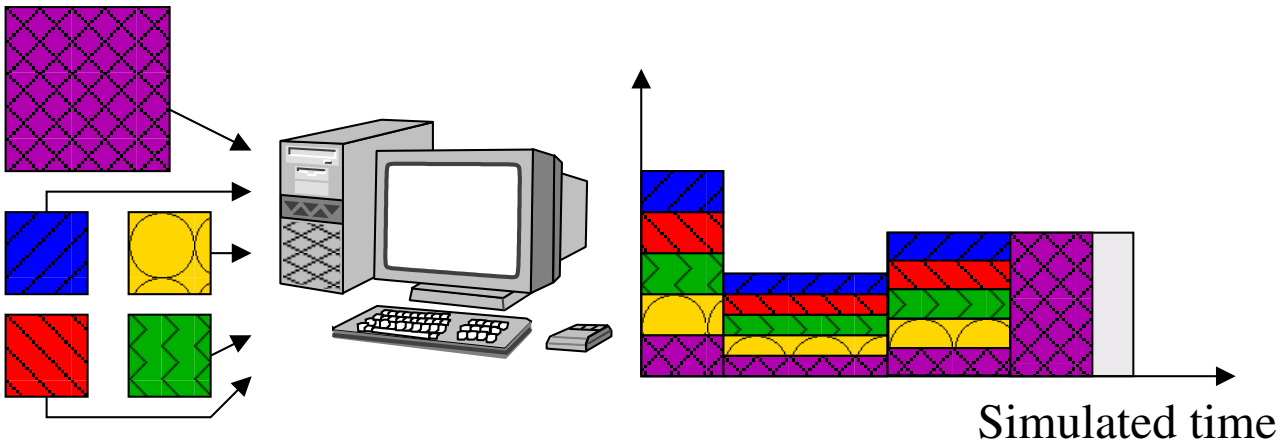
Adding Dynamic Availabilities to the Picture

Trace definition

- List of discrete events where the maximal availability changes
- $t_0 \rightarrow 100\%$, $t_1 \rightarrow 50\%$, $t_2 \rightarrow 80\%$, *etc.*

Adding traces doesn't change kernel main loop

- Availability changes: simulation events, just like action ends



SimGrid also accept state changes (on/off)

SURF main function

double surf_solve(void)

```
/* Search next action to end (look for min date over all models) */
xbt_dynar_foreach(model_list, iter, model) { /* check on model "model->name" */
  model_next_action_end = model->model_private->share_resources(NOW);
  if (min < 0.0 || model_next_action_end < min)
    min = model_next_action_end;
}
if (min < 0.0) return -1.0; /* no planned action end => simulation's over */

/* Handle every events occurring before min */
while ((next_event_date = tmgr_history_next_date(history)) != -1.0) {
  if (next_event_date > NOW + min) break; /* no further event before min */
  /* apply event by updating models */
  while((evt=tmgr_history_get_next_event_leq(history, next_event_date, &value, &resource))){
    if (resource->model->model_private->resource_used(resource))
      min = next_event_date - NOW; /* evt changes a resource currently used. Change min */

    /* update state of the model according to event */
    resource->model->model_private->update_resource_state(resource, evt, value, NOW + min);
  }
}
NOW = NOW + min; /* Increase the simulation clock (NOW is returned by SURF_get_clock() ) */

/* Ask models to update the state of actions they are responsible for according to the clock *
xbt_dynar_foreach(model_list, iter, model)
  model->model_private->update_actions_state(NOW, min);
```

recall

recall

More on model implementations

All the solving logic of models lies in 4 functions

`share_resource`: compute the sharing between actions on every resource

`resource_used`: whether action actively using the resource (sleep ones never do)

`update_resource_state`: apply trace events

`update_action_state`: reduce actions needs by what they just got

the hard thing is about sharing

Most models use a Linear MaxMin solver (lmm) to compute sharing

- Other binds to external tool (gtnets), or have no sharing (constant, timer)
- Comes down to a linear system where actions are variable and resource constants
- Disclamer: I only partially understand lmm internals for now

Actions and Resources

lmm resources

- `lmm_constraint` representing that resource in the system
- state (on/off) & handler of next state trace event
- power (+latency for network) & handler of next power trace event

lmm actions

- `lmm_variable` allowing the system to return the share gained by that action
- boolean indicating whether it's currently suspended

But you are free to do otherwise

- Example: constant network
  - When you start a communication, it will last a constant amount of seconds
  - No sharing, so no need for link resource at all!
  - `update_action_state` simply deduce the time delta to each remaining time
- Other example: GTNetS
  - Simply wrap the calls to the external solver
  - No internal intelligence

Parsing platform files (instantiating the resources)

We use FleXML as XML parser generator

- Write a flex file from DTD, converted in fast, dependence-free C parser
- Drawback: cannot cope with stuff not in the DTD

Generated parser are SAX-oriented (not DOM-oriented)

- You add callbacks to event lists. Naming schema:
  - <*tag*> $\rightsquigarrow$ `STag_surfxml_`*tag*`_cb_list`; </*tag*> $\rightsquigarrow$ `ETag_surfxml_`*tag*`_cb_list`
  - Ex: <host> $\rightsquigarrow$ `STag_surfxml_host_cb_list`;
- This is done (usually during init) with `surfxml_add_callback`
  Ex: `surfxml_add_callback(STag_surfxml_host_cb_list, &parse_host)`
- Attributes accessible through globals: `A_surfxml_`*tag*`_`*attrname*
  Ex: `host->power = sscanf("%f",A_surfxml_host_power)`

Using such callbacks, models should:

- Parse their resource and store them in `model.resource_set` dictionary
- But that's optional: Cste network ignore links; GTNetS don't store them itself

Writing your own model

What you should write (recap)

- Action and resource datastructures
- Model's methods:
  - Actions' constructors (depending on whether a network or cpu model)
  - Actions' getter/setters
  - The 4 methods of the sharing logic
  - Finalize
- Parser callbacks
- Plug your model in `surf_config.c` so that users can select it from cmd line
- Possibly update the DTD to add the new info you need at instanciation

Guidelines

- Reusing the existing is perfectly fine
  (everything is in there for lmm based models – more to come)
- Please do not *dupplicate* code (as too often done till now)
- (at least if you want to get your code integrated in the SVN)

Object Orientation of Actions and Resources

```
typedef struct {
  surf_model_t model;
  char *name;
  xbt_dict_t properties;
} s_surf_resource_t, *surf_resource_t;

typedef struct {
  s_surf_resource_lmm_t lmm_resource;

  double lat_current;
  tmgr_trace_event_t lat_event;
} s_link_CM02_t, *link_CM02_t;
```

```
typedef struct {
  double current;
  double max;
  tmgr_trace_event_t event;
} s_surf_metric_t;
typedef struct {
  s_surf_resource_t generic_resource;
  lmm_constraint_t constraint;
  e_surf_resource_state_t state_current;
  tmgr_trace_event_t state_event;
  s_surf_metric_t power;
} s_surf_resource_lmm_t, *surf_resource_lmm_t;
```

- link_CM02_t are pointers to struct
  (like every SimGrid datatype ending with _t and not beginning with s_)
- They can be casted to lmm resources or generic ones
  $\Rightarrow$ we can reuse generic implementation of services

Warning: there is **no** security here

More on prehistoric OOP in C (purists, please forgive)

```
typedef struct {
  surf_model_t model;
  char *name;
  xbt_dict_t properties;
} s_surf_resource_t, *surf_resource_t;

typedef struct {
  s_surf_resource_lmm_t lmm_resource;

  double lat_current;
  tmgr_trace_event_t lat_event;
} s_link_CM02_t, *link_CM02_t;
```

```
typedef struct {
  double current;
  double max;
  tmgr_trace_event_t event;
} s_surf_metric_t;
typedef struct {
  s_surf_resource_t generic_resource;
  lmm_constraint_t constraint;
  e_surf_resource_state_t state_current;
  tmgr_trace_event_t state_event;
  s_surf_metric_t power;
} s_surf_resource_lmm_t, *surf_resource_lmm_t;
```

```
surf_resource_t surf_resource_new(size_t childsize, surf_model_t model, char *name, xbt_dict_t p
  surf_resource_t res = xbt_malloc0(childsize);
  res->model = [...]
  return res;
}
surf_resource_lmm_t surf_resource_lmm_new(size_t childsize,
    /* for superclass */ surf_model_t model, char *name, xbt_dict_t props, /* our args */ [...])

  surf_resource_lmm_t res = (surf_resource_lmm_t)surf_resource_new(childsize,model,name,props);
  res->constraint = [...]
  return res;
}
link_CM02_t CM02_link_new([...]) {
  link_CM02_t res = (link_CM02_t) surf_resource_lmm_new(sizeof(s_link_CM02_t), [...]);
  [...]
```

Adding new kind of models

Motivation

- Disk is still missing in SimGrid
- You may want to model memory (although that's probably a bad idea)
- Everybody loves getting higher in abstraction (at least at university)

Sorry no easy path for that

What's to do?

- Add a new specific section in surf_model_t (easy)
- Update the workstation model (because it aggregates other models – quite easy)
  careful, there is 2 workstation models: raw aggregator and L07 for parallel tasks
- Allow upper lever to create specific actions to your model kind
  - That will be quite difficult and very long
  - You need to change almost everything (at least augment it)
    `MSG_task_disk(disk, 50)`
    ⇝ `SIMIX_action_create_diskuse(...)`
    ⇝ surf_model_disk->extension.disk.new_disk()
- We may be able to ease this task too (but is there a real need?)

Conclusion on SURF internals

Big picture

- Resources (XML created) provide power to actions (upper layer created)
- Design goal include effectiveness and extensibility (conceptual purity? Erm, no)

Extending it

- It is fairly easy to add new models for existing resource kinds
- It is quite long (and somehow difficult) to add new resource kinds

Future work

- Ongoing cleanups not completely finished
  - Still some dupplicated code
  - Resource power not handled consistantly accros models
- New models:
  - Highly scalable ones in USS-SimGrid project
  - Compound ones where CPU load reduce communication abilities
  - Multi-core
  - (pick your favorite one)

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
    - Big picture
  - Global Elements
  - Simix Process

Simix

SimIX provides a posix-like interface for writing user level APIs

- Virtualized processes that runs user level code
- Virtualized hosts that run the processes
- Synchronization primitives that rely in the surf layer
  - mutex
  - condition variables

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Global Elements

Simix state is contained in the following global data structure:

```
typedef struct SIMIX_Global {
  smx_context_factory_t context_factory;
  xbt_dict_t host;
  xbt_swag_t process_to_run;
  xbt_swag_t process_list;
  xbt_swag_t process_to_destroy;
  smx_process_t current_process;
  smx_process_t maestro_process;
  ...
};
```

Simix Main Loop

Simix main loop

- a processes (from process to run list)
- Execute it
-

Agenda

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work
- SimGrid Internals
  - SURF
  - Simix
  - Global Elements
  - Simix Process

Simix Process 1

The process is the a central element in Simix.
It is represented by the following datastructure:

```
struct s_smx_process {
  ...
  char *name;
  smx_host_t smx_host;
  smx_context_t context;
  ex_ctx_t *exception;
  int blocked : 1;
  int suspended : 1;
  int iwannadie : 1;
  smx_mutex_t mutex;
  smx_cond_t cond;
  xbt_dict_t properties;
  void *data;
};
```

Simix Process 2

Simix keeps for each process:

- an execution context
- an exception container
- its running state (blocked, suspended, killed)
- pointers to the mutex or condition vars where the process is waiting
- user level data provided by the user

Context

The Simix contexts are an abstraction of the execution state of a process plus an interface for controlling them.

Each context is composed of:

- a pointer to the code (main function of the process associated)
- an execution stack
- the state of the registers
- some functions for scheduling/unscheduling the context

There are 3 implementations of the context interface (factories)

- a pthread based one
- a ucontext based one
- a java based one (uses jprocess)

- Model-Checking within SimGrid
  - Introduction to Model-Checking
  - Adding Model-Checking to SimGrid
  - Current Status and Future Work

- SimGrid Internals
  - SURF
    - Big Picture
    - Models
    - How Models get used
    - Actions and Resources
    - Writing your own model
    - Adding new kind of models
  - Simix
    - Big picture
  - Global Elements
  - Simix Process