GRAS: a Research and Development Framework for Grid and P2P Infrastructures

Martin Quinson ⟨martin.quinson@loria.fr⟩

Nancy University / LORIA (France)

Parallel and Distributed Computing and Systems (PDCS 2006)
November 13, 2006

Context

Modern computational platforms

- Grid and P2P systems aggregate distributed resources

☺ Almost infinite potential ☹ Difficult to use

Main characteristic: large scale

- Range from a few dozen nodes to millions

⇒ Heterogeneous: hardware, software, even administrative orientations

⇒ Dynamic: quantitative (bandwidth variation) and qualitative (resource churn)

Difficulties

- Theoretical: Heuristics mandatory for NP-hard problems (scheduling, routing, etc)
  But dynamicity hinders experiment reproductability!
  ⇒ Temptation of a simulator, but we want more than prototyping
- Technical: Setup a development and experimentation environment difficult
  ⇒ Need for an adapted runtime environment

Motivation

Applications need to rely on infrastructures

A lot of middlewares and infrastructures exist already

- Remote execution (NetSolve, DIET, APST, Condor)
- Platform monitoring (NWS) and discovery (ENV)
- A bit of everything (Globus)

Infrastructures are themselves difficult to develop, assess and tune

- Can be seen as large-scaled distributed applications
- Several entities placed on nodes interacting with application level protocol
- Specificity: Distribution must not be masked

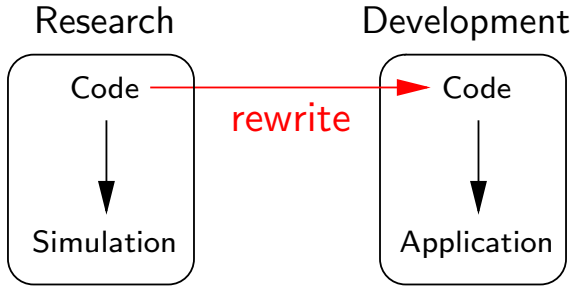
The GRAS project

- Aims at easing the development of network-aware applications
- Development on simulator, deployment without modification

Goals of the GRAS project

Easing infrastructure development

Development of real distributed applications using a simulator

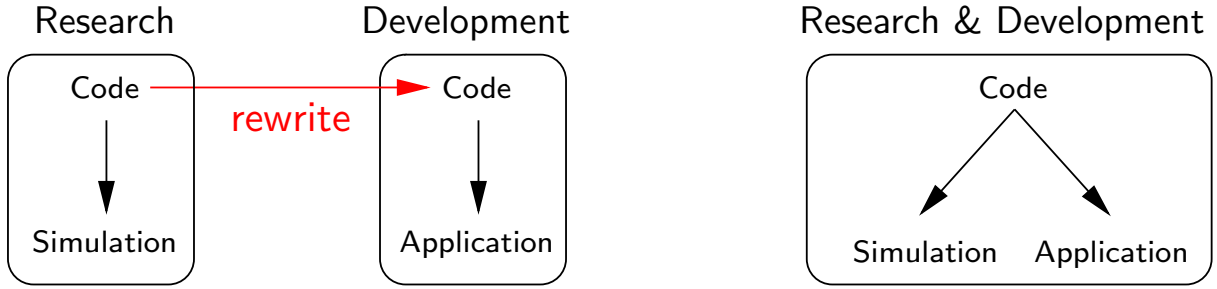


Without GRAS

Goals of the GRAS project

Easing infrastructure development

Development of real distributed applications using a simulator



Without GRAS

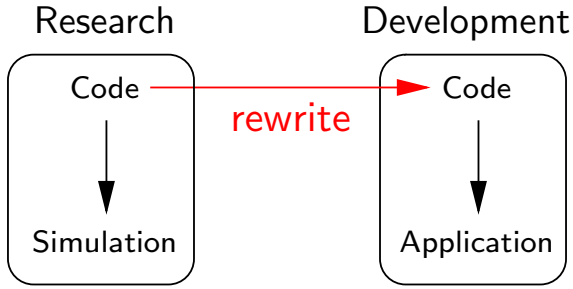
With GRAS

- Framework for Rapid Development of Distributed Infrastructure
  - Develop and tune on the simulator; Deploy *in situ* without modification

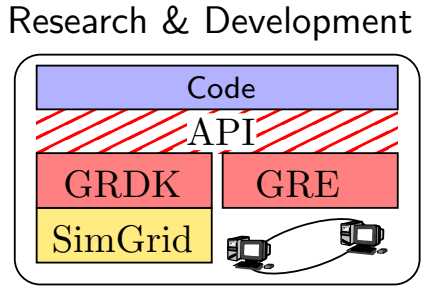
Goals of the GRAS project

Easing infrastructure development

Development of real distributed applications using a simulator



Without GRAS



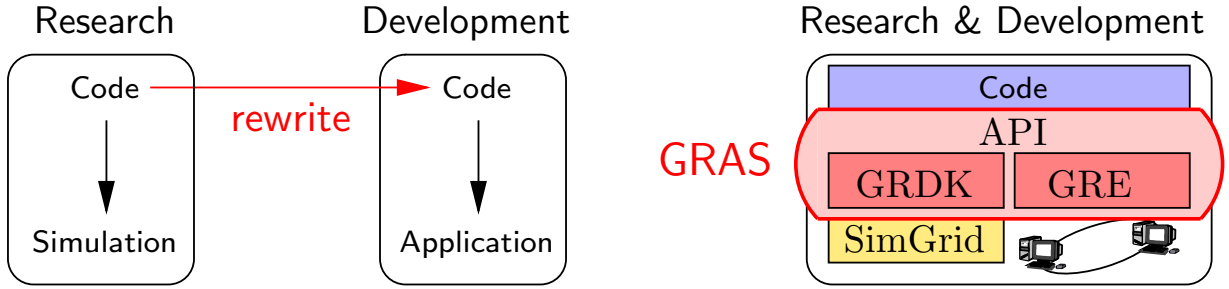
With GRAS

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

Goals of the GRAS project

Easing infrastructure development

Development of real distributed applications using a simulator

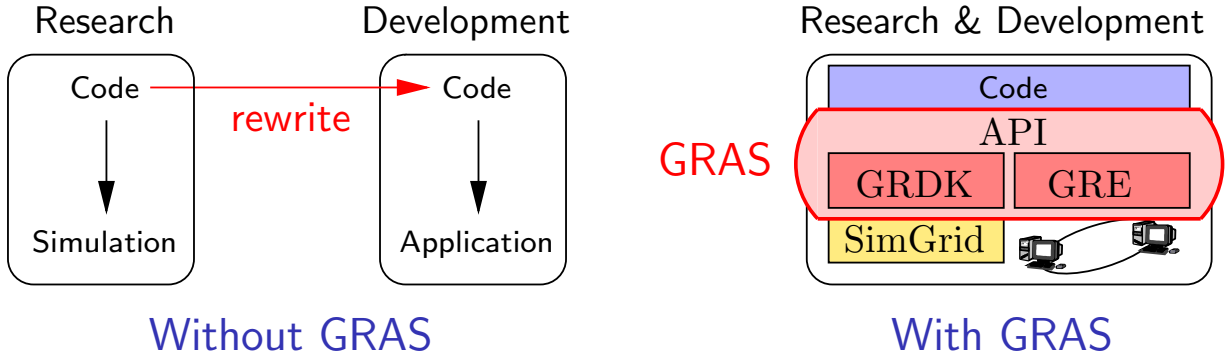


- Framework for Rapid Development of Distributed Infrastructure
  - Develop and tune on the simulator; Deploy *in situ* without modification
    How: One API, two implementations

Goals of the GRAS project

Easing infrastructure development

Development of real distributed applications using a simulator

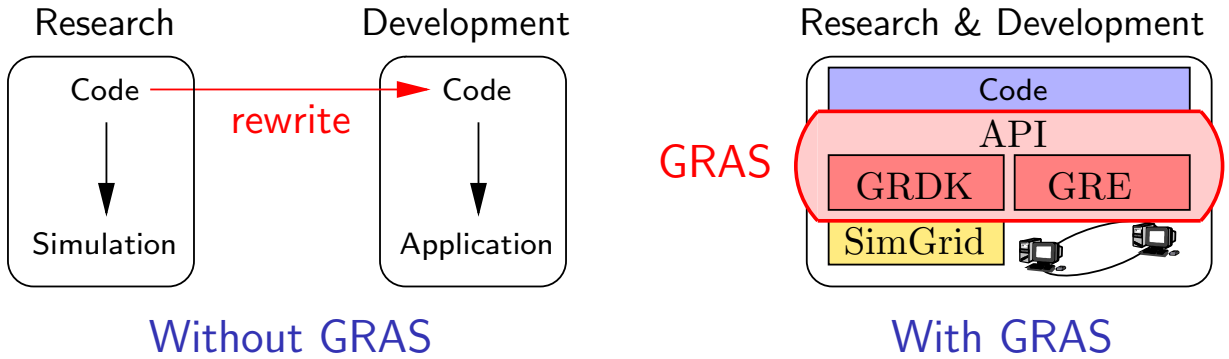


- Framework for Rapid Development of Distributed Infrastructure
  - Develop and tune on the simulator; Deploy *in situ* without modification
    How: One API, two implementations
- Efficient Grid Runtime Environment (result = application $\neq$ prototype)

Goals of the GRAS project

Easing infrastructure development

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

Presentation outline

The simulator used: SimGrid [Casanova, Legrand]

Standard simulator for grid application studies

SimGrid functionalities

- Complex and realistic platforms (with availability variations and resource churn)
- Based on fluid models (unlike packet based network simulator like NS2)
  - Fast: simulates several hours per second
  - Thousands of simulated processes on a single host
  - Satisfying (ongoing) validation

GRAS and SimGrid

- GRAS hides SimGrid details: applications just "work" on the simulator
- Actually, GRAS is now part of the SimGrid toolbox

Main concepts of the GRAS API

Agents (acting entities)

Messages (what gets exchanged between agents)

Callbacks (code to execute when a message is received)

Main concepts of the GRAS API

Agents (acting entities)

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

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

Conveying structured data

GRAS message payload can be any valid C type

- Structure, enumeration, array, pointer, ...
- Classical garbage collection algorithm to deep-copy it

Describing a data type to GRAS

- Can be manually described, or automatically parsed from C type declaration

GRAS wire protocol: NDR (Native Data Representation)

Avoid data conversion when possible:

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

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 current time? How to get the process sleeping?
  - 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

Presentation outline

Assessing communication performance

Only communication performance studied since computation are not mediated

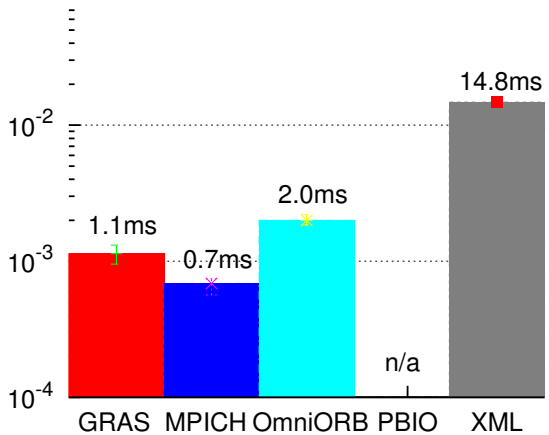
- Experiment: timing ping-pong of structured data (a message of Pastry)

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

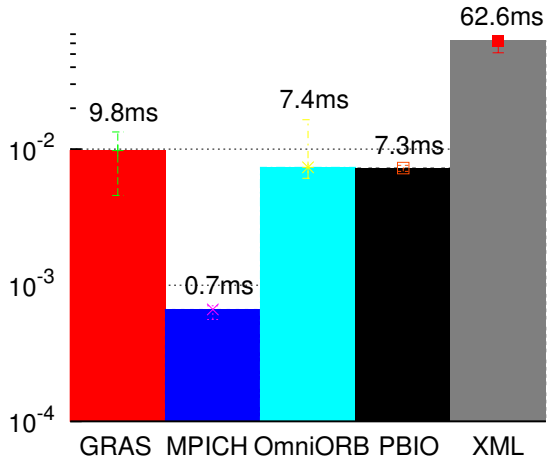
```
typedef struct {
  int which_row;
  int row[COLS][MAX_ROUTESET];
} row_t;
```

- Tested solutions
  - GRAS
  - PBIO (uses NDR)
  - OmniORB (classical CORBA solution)
  - MPICH (classical MPI solution)
  - XML (Expat parser + handcrafted communication)
- Platform
  - Scale: intra-machine; on LAN; on WAN
  - Architectures: x86, PPC, sparc (all under Linux)

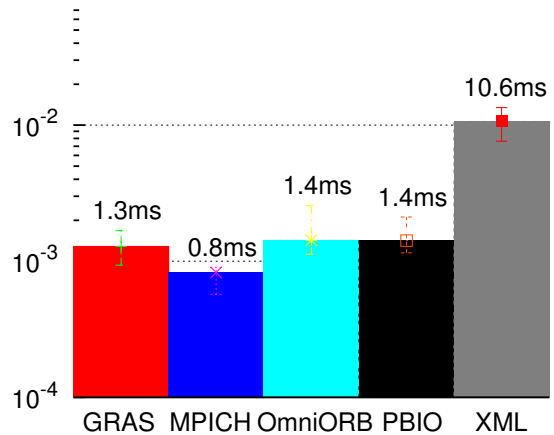
Intra-machine results



PPC



SPARC

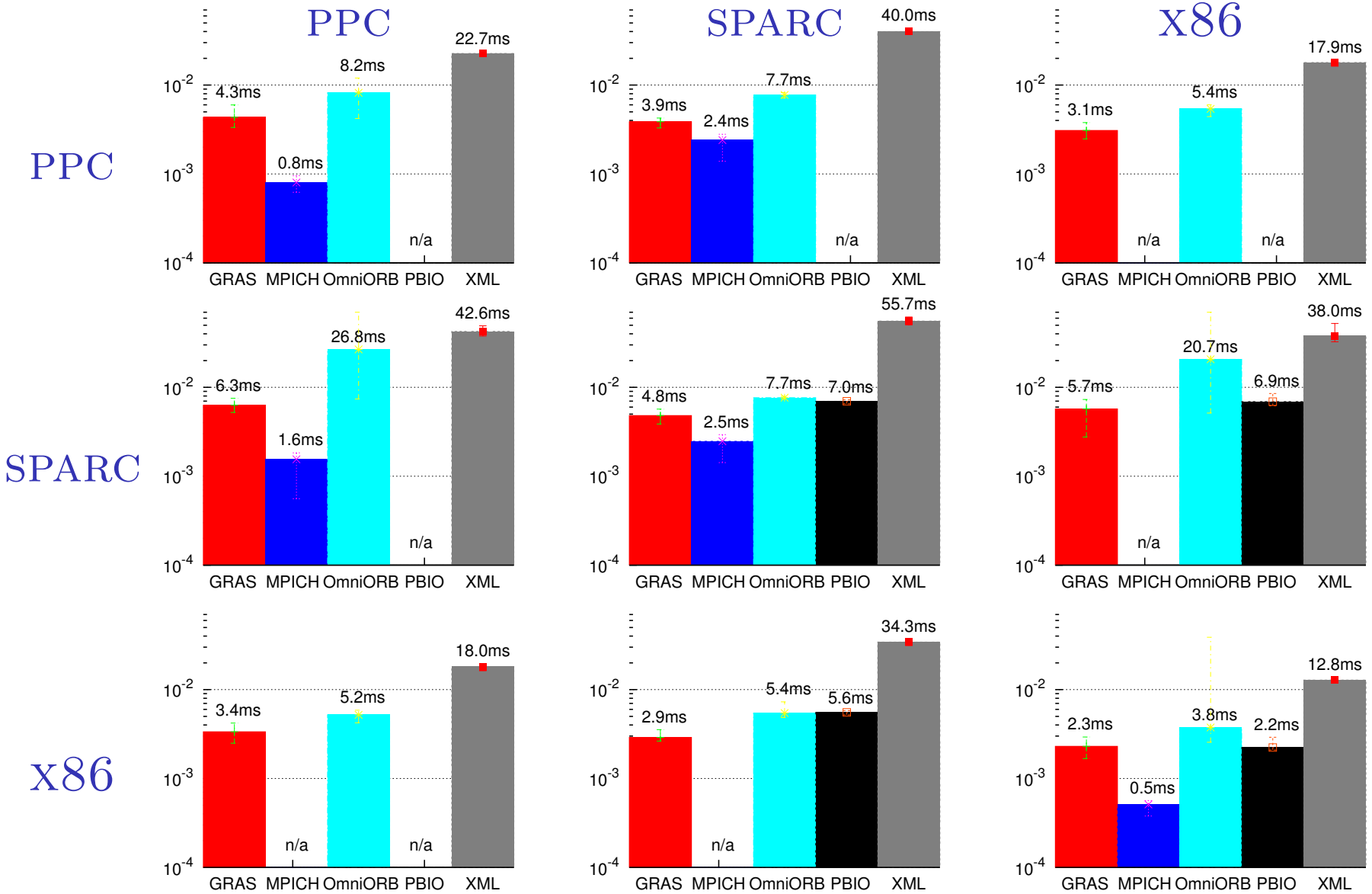


x86

Discussion

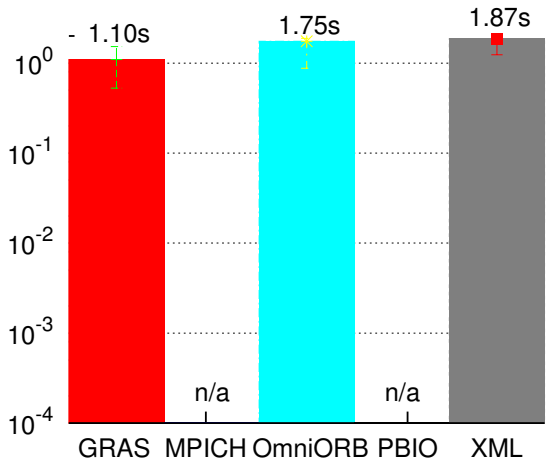
- Portability: PBIO broken on PPC
- Performance: XML ways slower (extra conversions + verbose wire encoding)

LAN results (column: sender; row: receiver)

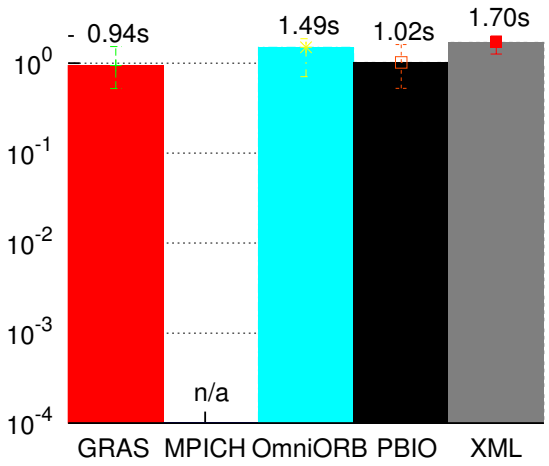


- MPICH twice as fast as GRAS, but cannot mix little- and big-endian Linux
- GRAS second best solution

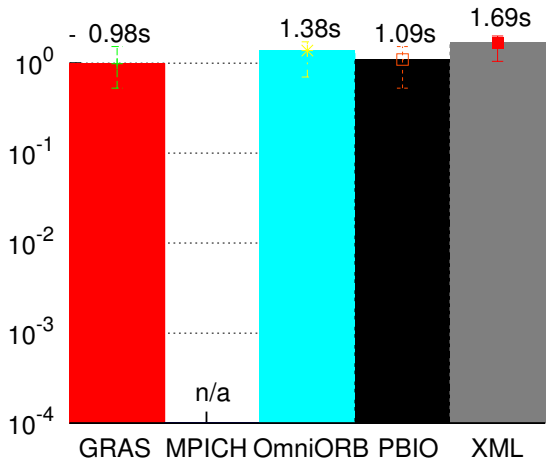
WAN Results (California → France)



PPC



SPARC



x86

- Less performance difference on WAN
- Portability: failed to use MPICH on WAN

Experiment conclusion

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

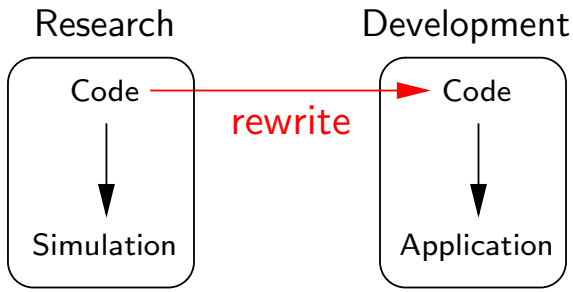
- XML complexity may be artefact of Expat parser (but fastest)
- MPICH: manual marshaling/unmarshalling
- PBIO: automatic marshaling, but manual type description
- OmniORB: automatic marshaling, IDL as type description
- GRAS: automatic marshaling & type description (IDL is C)

Conclusion

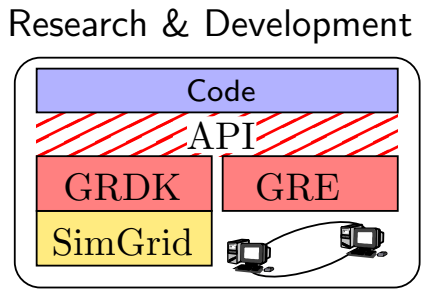
GRAS is the less demanding solution from developer perspective

Presentation outline

Conclusion: GRAS eases infrastructure development



Without GRAS



With GRAS

GRDK: Grid Research & Development Kit

- API for (explicitly) distributed applications
- Uses a fast yet realistic simulator (SimGrid)

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
  - Easy to deploy: C ANSI; no dependency; autotools; <400kb
  - Extensive toolbox: data containers, logs, config, exceptions, test units, etc.

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

Ongoing applications

- Comparison of P2P protocols (Pastry, Chord, etc)
- Network mapper (ALNeM): capture platform descriptions for simulator
- Large scale mutual exclusion service

SimGrid Distribution

- LGPL, 30 000 lines of code
- http://gforge.inria.fr/projects/simgrid/
- Examples, documentation and tutorials on the web page

Thanks for attending

Any question?

Presentation outline

- Introduction
- The GRAS project
  - Project goals
  - The SimGrid simulator
  - Offered API
  - Efficient communication of structured data
  - Emulation and Virtualization
- Experimental evaluation
  - Assessing communication performance
  - Assessing API simplicity
- Conclusion and Perspectives

Appendix

- Example of code: Ping-Pong
- Describing data types to GRAS
- Network simulators vs SimGrid
- Simulation kernel details
- Details on the emulation support
- Visualizing the simulation
- Model-checking

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

Common to client and server

```
#include "gras.h"
XBT_LOG_NEW_DEFAULT_CATEGORY(Ping,"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, gras_msgtype_by_name("ping"), &ping);
  INFO3("PING(%d) -> %s:%d",ping, gras_socket_peer_name(peer), gras_socket_peer_port(peer));
  gras_msg_wait(6000,gras_msgtype_by_name("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(gras_msgtype_by_name("ping"),&server_cb_ping_handler);

  gras_msg_handle(600.0);
  if (!globals->endcondition) WARN0("An error occured: endcondition != 0");

  free(globals);  gras_exit();  return 0;
}
static int server_cb_ping_handler(gras_socket_t expeditor, void *payload_data) {
  int msg=*(int*)payload_data;                                  /* Get the payload */
  server_data_t *globals=(server_data_t*)gras_userdata_get(); /* Get the globals */

  /* Send data back as payload of a pong message to the ping's expeditor */
  gras_msg_send(expeditor, gras_msgtype_by_name("pong"), &msg);

  globals->endcondition = 1;
}
```

Conveying structured data (details)

Manual description (excerpt)

```
gras_datadesc_type_t gras_datadesc_struct(const char *name);
void gras_datadesc_struct_append(gras_datadesc_type_t s, char*name, gras_datadesc_type_t field);
void gras_datadesc_struct_close(gras_datadesc_t struct_type);
```

Automatic description of a classical matrix type

```
GRAS_DEFINE_TYPE(s_matrix,
  struct s_matrix {
    int rows; int cols;
    double *data GRAS_ANNOTE(size,rows*cols);
  }
);
```

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

Existing wire protocols

- XDR Everything converted to a "common language"
- CDR Scalar converted on need only; Structured data converted anyway
- NDR Scalar and structures converted on need only (used by GRAS)

Network and Grid Simulators

Usual Network Simulators

- Goals:
  - Understand networks behavior, routing protocols, QoS, ...
  - Identify and overcome limitations of network protocols

  $\Rightarrow$ Precise simulation of the packet movements along links
- Examples: NS, DaSSF, OMNeT++

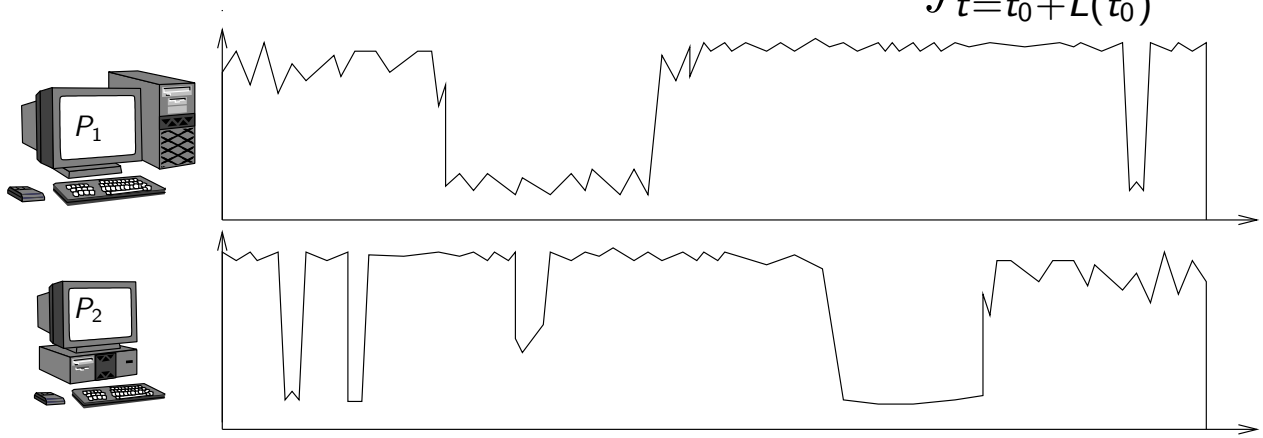
Our needs:

- Network behavior as experienced by applications: no need for packets
- Something fast (during debugging phase): no need for all details
- Must consider CPU resource

Solution: SimGrid [Casanova, Legrand]

Under the hood: the simulation kernel

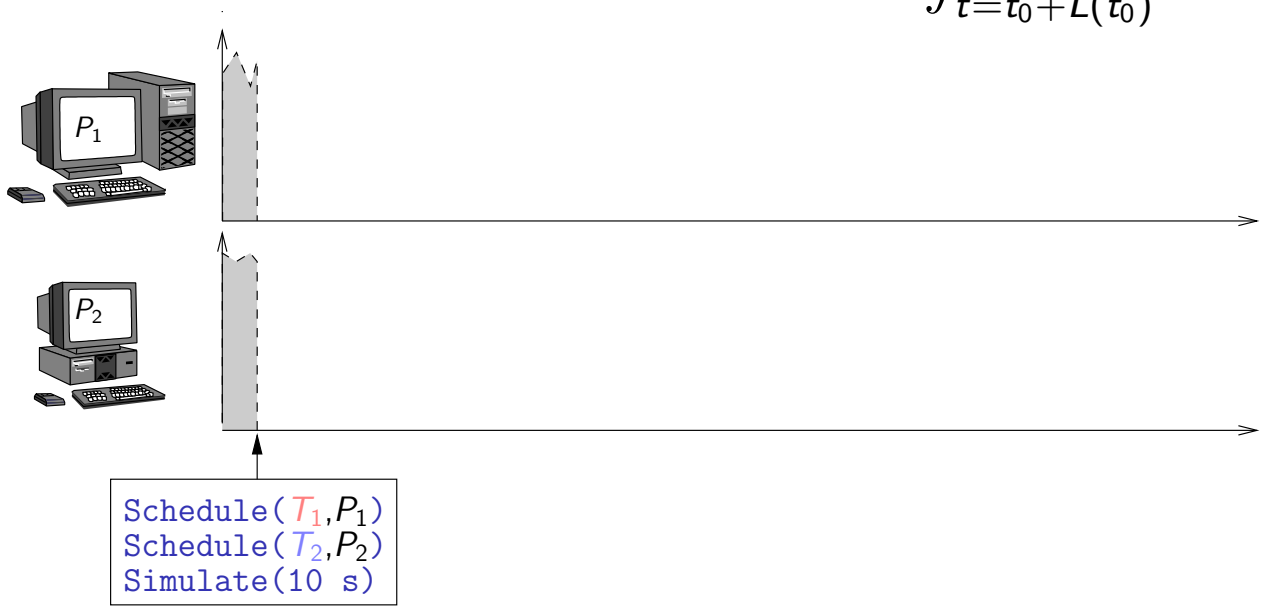
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

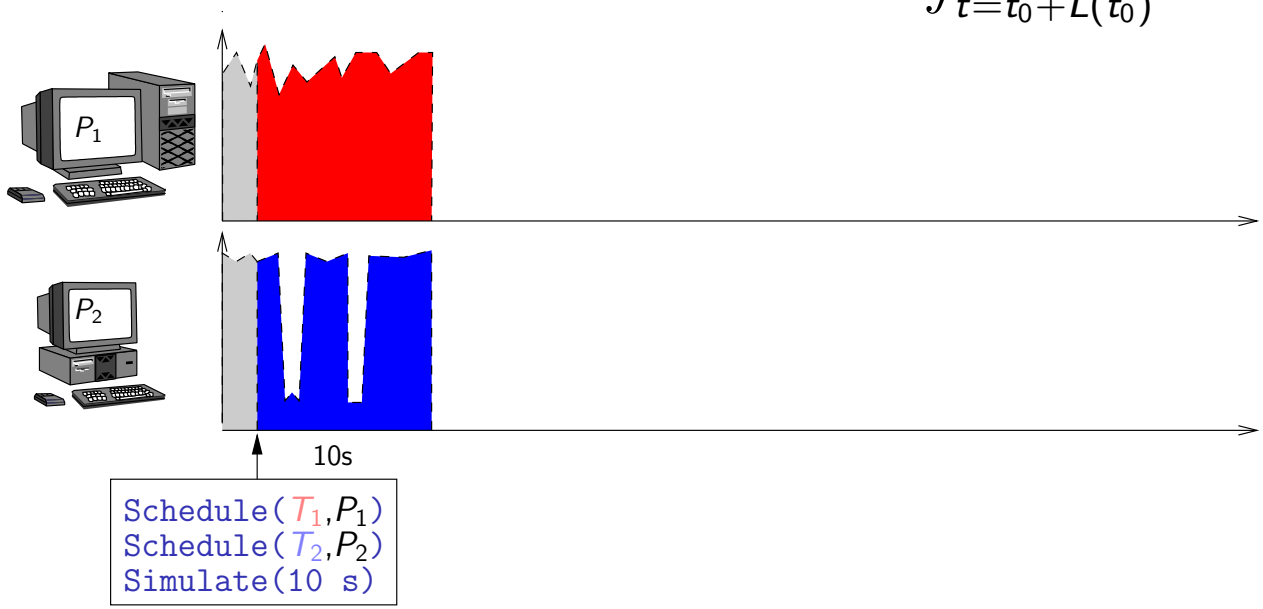
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

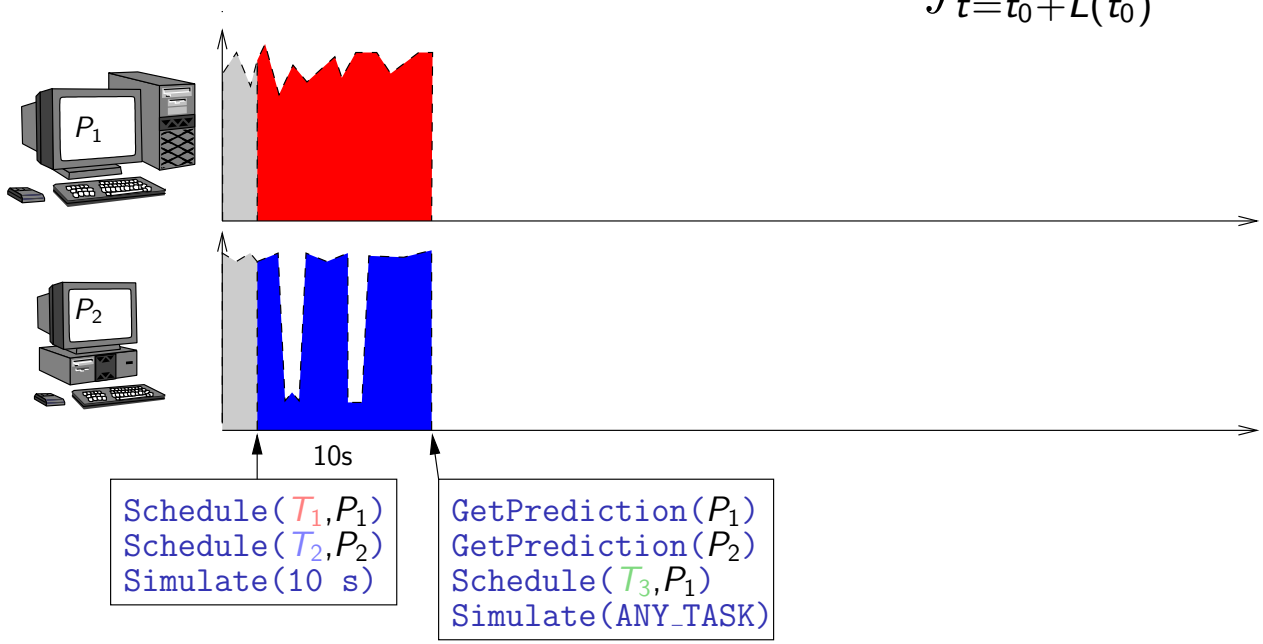
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

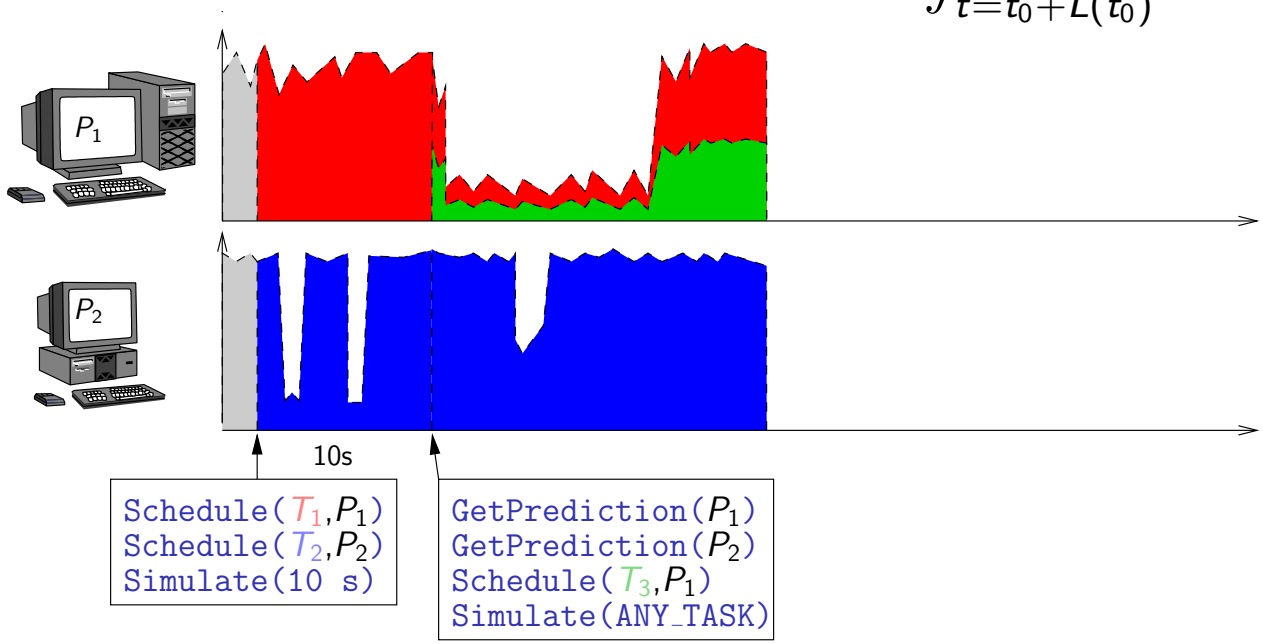
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

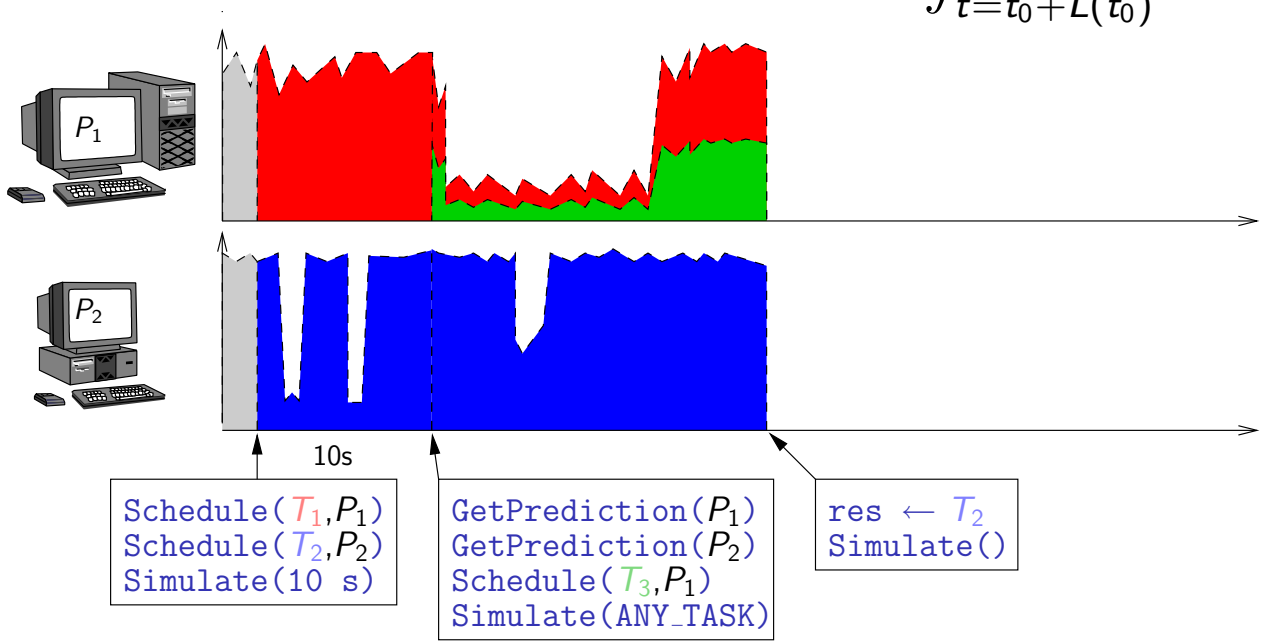
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

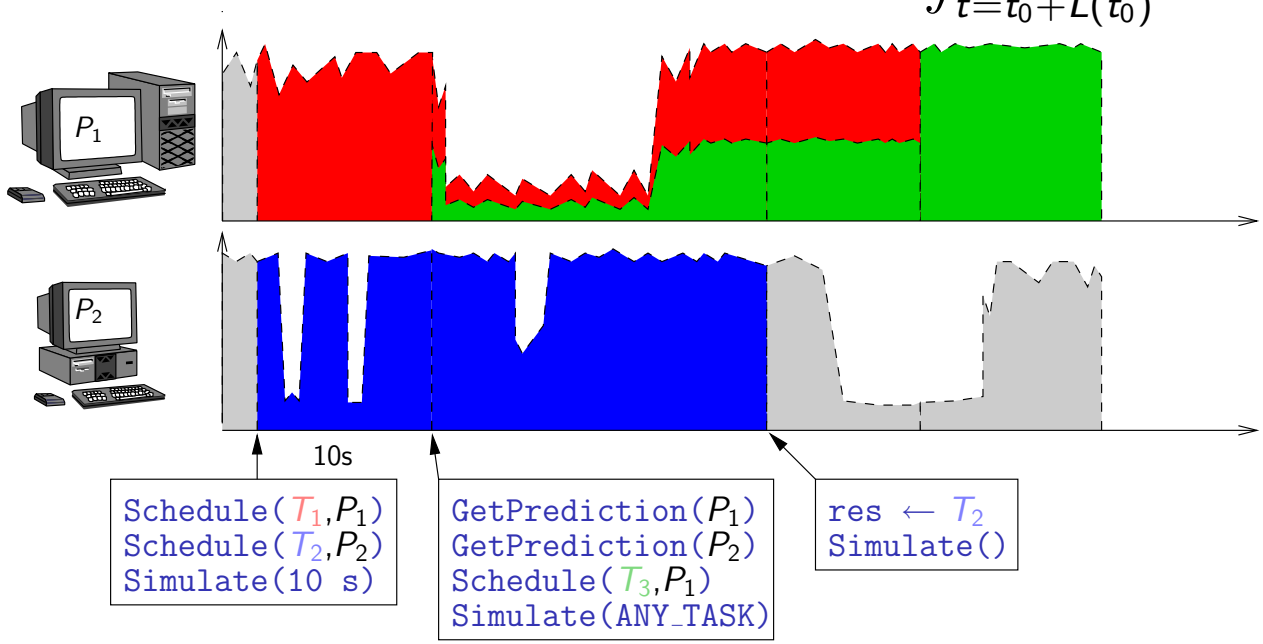
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

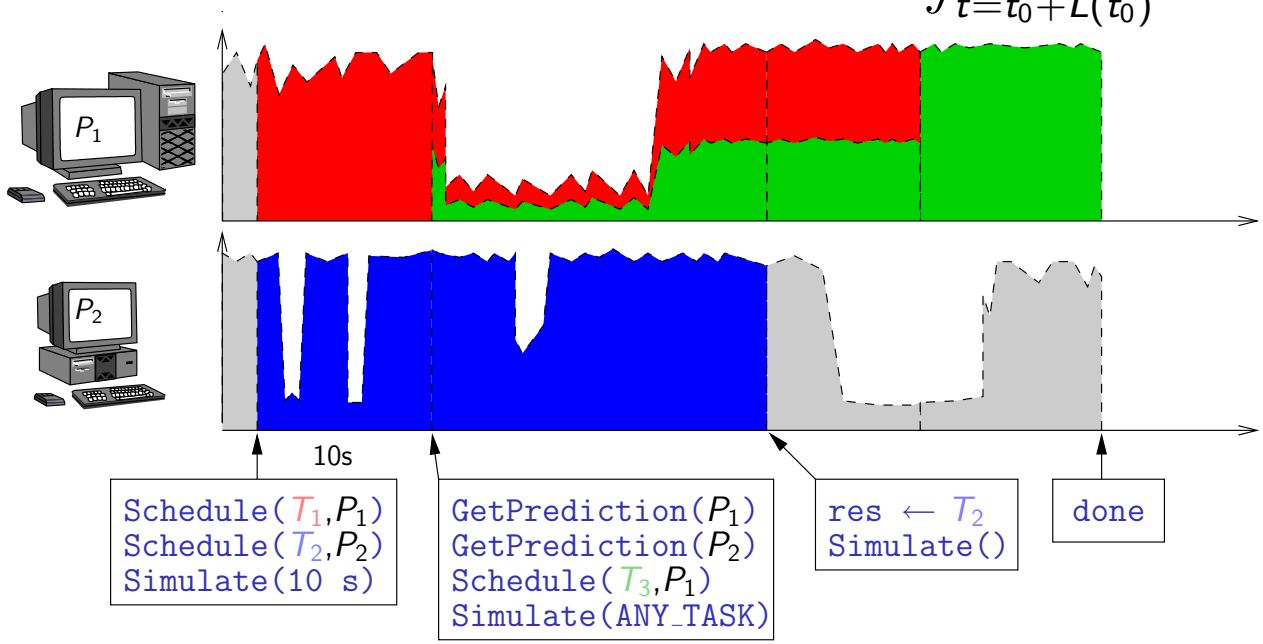
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: the simulation kernel

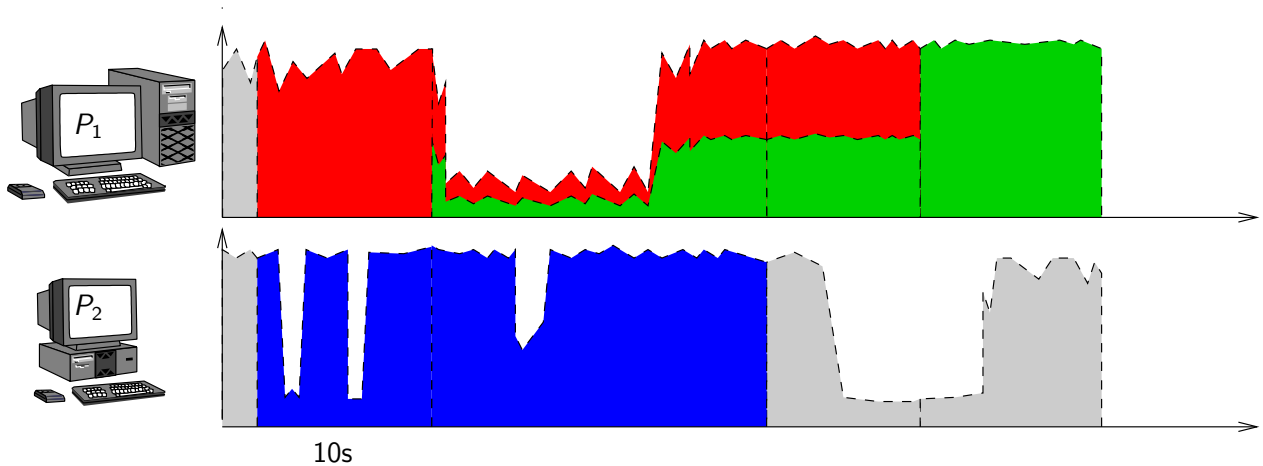
- Main objects:
  - Resource: Availability trace (CPU, BW) + Latency trace + Sharing policy
  - Task: Amount of work
- Tasks are scheduled onto resources
- Simulator computes task ending time T: $Work = \int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt$
- Example:



- Sharing policies: Sequential, Shared or TCP
  TCP sharing modeled as Max-Min fairness (proportional soon

Under the hood: implementing multiple agents

- Several execution contexts:
  - One context per agent
  - A maestro context
- Control flow
  - Agent runs `gras_msg_send()` and similar ⇒ control passed to maestro
  - Maestro *schedules* the task within simulation kernel
  - Maestro asks simulation kernel which scheduled tasks are done
  - Maestro passes control to corresponding agent



- Implementation: pthreads or ucontexts (lighter, faster but UNIX98)

  Some hundreds or thousands of simulated processes on a single host

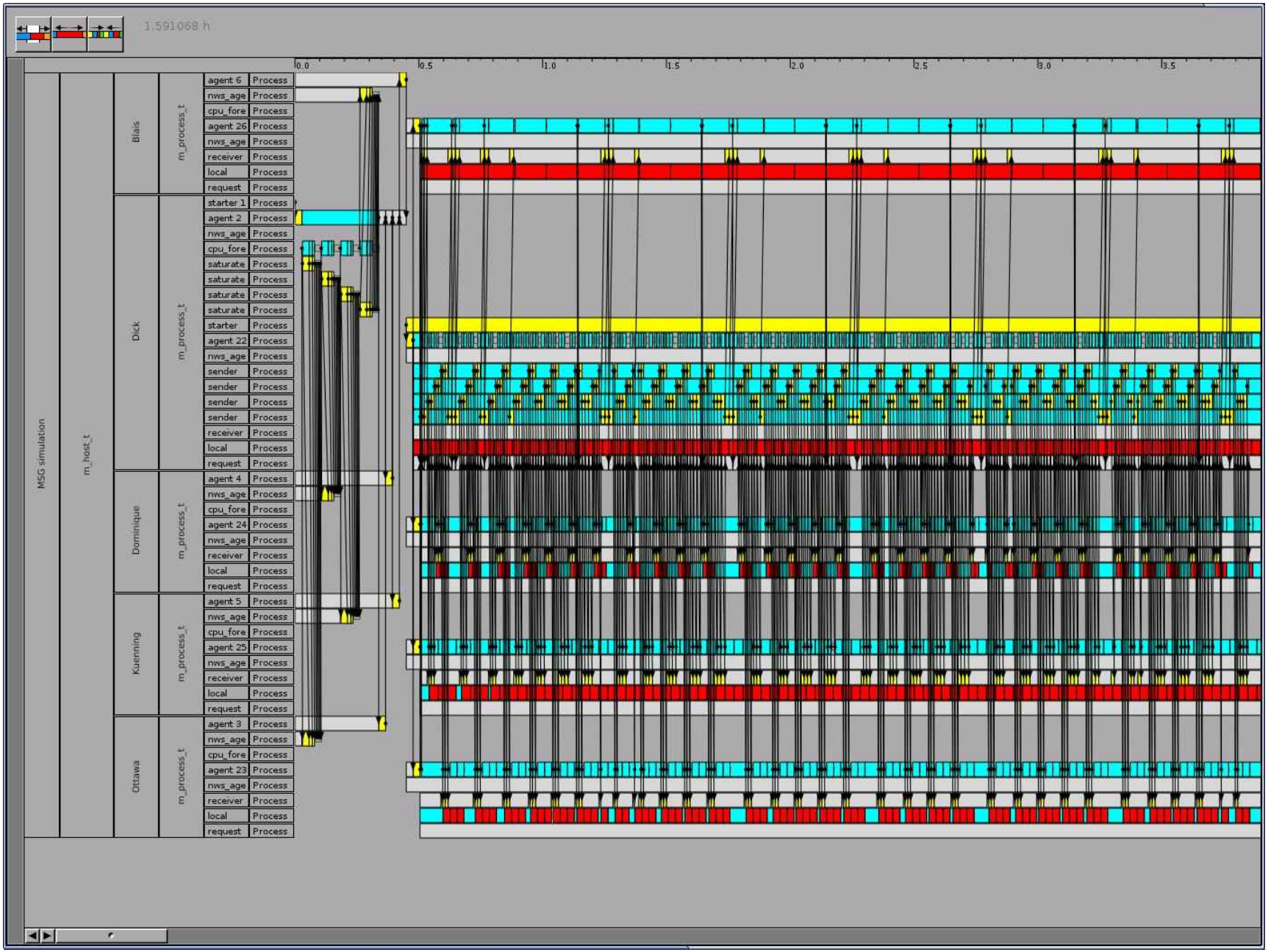
Details on the emulation support

- We have a collection of macro to automatically report computation time into the simulator:

	Run on host?	Bench'ed?	Time reported?
GRAS_BENCH_ALWAYS_BEGIN() GRAS_BENCH_ALWAYS_END()	Each time	Each time	Each time
GRAS_BENCH_ONCE_RUN_ONCE_BEGIN() GRAS_BENCH_ONCE_RUN_ONCE_END()	First time	First time	Each time
GRAS_BENCH_ONCE_RUN_ALWAYS_BEGIN() GRAS_BENCH_ONCE_RUN_ALWAYS_END()	Each time	First time	Each time

- Other problems to solve:
  - What about global data?
    - Agent status placed in a specific structure, ad-hoc manipulation API
  - How to write the `main()`?
    - Use another name (as usual with threads, the real `main()` is generated)

Visualizing the simulation: SimGrid generates Pajé traces



Model-checking GRAS application (still to be done)

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 $\longrightarrow$ conversion to C difficult
  - Test some cases $\longrightarrow$ 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)
  We know how to marshal C datatypes
- Second difficulty: Fight against combinatorial explosion
  Simulation can easily be distributed (is it enough?)