

SimGrid 101

Getting Started with SimGrid

Da SimGrid Team

April 3, 2012



Outline

- Installing SimGrid
 - Stable release
 - Unstable Version
 - The Bindings
- Your First SimGrid Program
 - User Interface(s)
 - Master/Workers
 - Trace Replay
- Further topics
 - Configuring your simulators
 - Surviving in C
 - Bindings Performance
- Conclusion

Installing a stable version (most advised for users)

On Debian, Ubuntu and similar

- ▶ `sudo apt-get install simgrid`

On Windows

- ▶ Get the installer (from page below), execute it and follow the instructions

From the sources

1. Get the archive: (see below for URL)
2. Open it: `tar xzf simgrid-*.tar.gz`
3. Configure it: `cmake .` or `ccmake .`
4. Install it: `make install`

Download page of the project:

- ▶ Direct access: https://gforge.inria.fr/frs/?group_id=12
- ▶ Idem + more info: <http://simgrid.gforge.inria.fr/download.php>

Details: <http://simgrid.gforge.inria.fr/simgrid/<version>/doc/install.html>

Installing an unstable version (developers only!)

Is unstable for you?

- ▶ Simple Rule of Thumb:
 - ▶ You plan to **use** SimGrid \leadsto nope, play safe with stable
 - ▶ You plan to **improve** SimGrid \leadsto yes, use unstable
- ▶ The reason why we name it “unstable”: we didn’t test it on all platforms
- ▶ It *can* be relatively usable at a given time, but we cannot promise.
- ▶ It *may* fail strangely on you, too. You’re on your own here.

Actually installing unstable

- ▶ Get source from git:

```
git clone git://scm.gforge.inria.fr/simgrid/simgrid.git
```
- ▶ Configure and installing (see instructions for stable)

Build Dependencies

- ▶ Depending on what you’re touching, you may need more softwares:
 - ▶ If you change the XML parsers, you need both flexml and flex

The Bindings

Some people don't like coding in C

- ▶ That's reasonable since C is the modern assembly language:
It can reveal faster but rather verbose and really tedious to get right
- ▶ Using C is not enough for maximal performance: you need to really master it

Bindings available for: Java, lua and Ruby

- ▶ **Why Java:** Every potential intern knows it (I guess)
- ▶ **Why Lua:** As simple as script language, but as efficient as C
- ▶ **Why Ruby:** Our team counts very effective Ruby lobbyists
- ▶ “Will you add my favorite language?”
 - ▶ We could, but it's rather time consuming (threading mess, at least)
 - ▶ We probably won't do it ourselves (our time is limited); we welcome patches

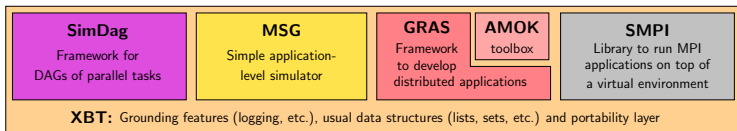
Installing the Bindings

- ▶ lua is included in the main archive, the others are separated
- ▶ Grab their archives, open it, read the README, build it, install it
- ▶ You need to install the main SimGrid archive to get the bindings working

Outline

- Installing SimGrid
 - Stable release
 - Unstable Version
 - The Bindings
- Your First SimGrid Program
 - User Interface(s)
 - Master/Workers
 - Trace Replay
- Further topics
 - Configuring your simulators
 - Surviving in C
 - Bindings Performance
- Conclusion

User-visible SimGrid Components



SimGrid user APIs

- ▶ **SimDag**: specify heuristics as DAG of (parallel) tasks
- ▶ **MSG**: specify heuristics as Concurrent Sequential Processes (Java/Ruby/Lua bindings available)
- ▶ **GRAS**: develop real applications, studied and debugged in simulator
- ▶ **SMPI**: simulate MPI codes

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 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: **MSG** (by far)

The MSG User Interface

Main MSG abstractions

- ▶ **Agent:** some code, some private data, running on a given host
- ▶ **Task:** amount of work to do and of data to exchange

- ▶ **Host:** location on which agents execute
- ▶ **Mailbox:** Rendez-vous points between agents (think of MPI tags)
 - ▶ You send stuff to a mailbox; you receive stuff from a mailbox
 - ▶ Network location of sender & receiver have no impact on rendez-vous; Communication timings of course take these locations into account
 - ▶ Mailboxes' identifiers are strings, making user code *ways* easier (either `host:port`, yellow page mechanism or whatever you want)

More information

- ▶ `examples/msg` in archive; Reference doc: `doc/group__MSG__API.html`
- ▶ Interface extended, never modified since 2002 (if using `MSG_USE_DEPRECATED`)

The MSG User Interface

Main MSG abstractions

- ▶ **Agent:** some code, some private data, running on a given host
one function + arguments coming from deployment XML file
- ▶ **Task:** amount of work to do and of data to exchange
 - ▶ `MSG_task_create`(name, compute_duration, message_size, void *data)
 - ▶ **Communication:** `MSG_task_{send,recv}`, `MSG_task_Iprobe`
 - ▶ **Execution:** `MSG_task_execute`
`MSG_process_sleep`, `MSG_process_{suspend,resume}`
- ▶ **Host:** location on which agents execute
- ▶ **Mailbox:** Rendez-vous points between agents (think of MPI tags)
 - ▶ You send stuff to a mailbox; you receive stuff from a mailbox
 - ▶ Network location of sender & receiver have no impact on rendez-vous; Communication timings of course take these locations into account
 - ▶ Mailboxes' identifiers are strings, making user code ways easier (either `host:port`, yellow page mechanism or whatever you want)

More information

- ▶ `examples/msg` in archive; Reference doc: `doc/group__MSG__API.html`
- ▶ Interface extended, never modified since 2002 (if using `MSG_USE_DEPRECATED`)

Executive Summary (detailed below)

1. Write the Code of your Agents

```
int master(int argc, char **argv) {  
    for (i = 0; i < number_of_tasks; i++) {  
        t=MSG_task_create(name, comp_size, comm_size, data);  
        sprintf(mailbox, "worker-%d", i % workers_count);  
        MSG_task_send(t, mailbox);}  
}
```

```
int worker(int ,char**) {  
    sprintf(my_mailbox, "worker-%d", my_id);  
    while(1) {  
        MSG_task_receive(&task, my_mailbox);  
        MSG_task_execute(task);  
        MSG_task_destroy(task);}  
}
```

2. Describe your Experiment

XML Platform File

```
<?xml version='1.0'?>  
<!DOCTYPE platform SYSTEM  
    "http://simgrid.gforge.inria.fr/simgrid.dtd"  
<platform version="3">  
<AS routing="Full">  
    <host name="host1" power="1E8"/>  
    <host name="host2" power="1E8"/>  
    ...  
    <link name="link1" bandwidth="1E6"  
        latency="1E-2" />  
    ...  
    <route src="host1" dst="host2">  
        <link_ctn id="link1"/>  
    </route>  
</AS>  
</platform>
```

XML Deployment File

```
<?xml version='1.0'?>  
<!DOCTYPE platform SYSTEM  
    "http://simgrid.gforge.inria.fr/simgrid.dtd">  
<platform version="3">  
    <!-- The master process -->  
    <process host="host1" function="master">  
        <argument value="10"/><!--argv[1]:#tasks-->  
        <argument value="1"/><!--argv[2]:#workers-->  
    </process>  
  
    <!-- The workers -->  
    <process host="host2" function="worker">  
        <argument value="0"/></process>  
</platform>
```

3. Write a main gluing things together, link and run

Master/Workers: Describing the Agents (1/2)

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

```
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(buff, task_comp_size, task_comm_size, NULL);  
        sprintf(mailbox, "worker-%d", i % workers_count);  
        XBT_INFO("Sending %s" to mailbox %s", task->name, mailbox);  
        MSG_task_send(task, mailbox);  
    }  
  
    /* Send finalization message to workers */  
    XBT_INFO("All tasks dispatched. Let's stop workers");  
    for (i = 0; i < workers_count; i++) {  
        sprintf(mailbox, "worker-%d", i % workers_count);  
        MSG_task_send(MSG_task_create("finalize", 0, 0, 0), mailbox);  
    }  
  
    XBT_INFO("Goodbye now!"); return 0;  
}
```

Master/Workers: Describing the Agents (2/2)

```
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_assert(errcode == MSG_OK, "MSG_task_get failed");

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

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

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

Master/Workers: gluing things together

```
int main(int argc, char *argv[ ]) {  
  
    MSG_global_init(&argc,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();  
  
    XBT_INFO("Simulation took %g seconds",MSG_get_clock());  
}
```

Compiling and Executing the result

```
$ gcc *.c -lsimgrid -o my_simulator  
$ ./my_simulator platform.xml deployment.xml  
[verbose output removed]
```

Master/Workers: 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 "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">

  <!-- 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>
```

Thanks to mailboxes, the master doesn't have to know where the workers are (nor the contrary)

Master/Worker in Java (1/2)

```
import simgrid.msg.*;
public class BasicTask extends simgrid.msg.Task {
    public BasicTask(String name, double computeDuration, double messageSize) {
        super(name, computeDuration, messageSize);
    }
}

public class FinalizeTask extends simgrid.msg.Task {
    public FinalizeTask() {
        super("finalize",0,0);
    }
}

public class Worker extends simgrid.msg.Process {
    public void main(String[] args)
        throws TransferFailureException, HostFailureException,
        TimeoutException, TaskCancelledException {
        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!");
    }
}
```

Master/Workers in Java (2/2)

```
import simgrid.msg.*;
public class Master extends simgrid.msg.Process {
    public void main(String[] args) throws MsgException {
        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!");
    }
}
```

The rest of the story

- ▶ No need to write the glue (thanks to Java introspection)
- ▶ The XML files are exactly the same (beware of capitalization for deployment)
- ▶ Output very similar too

Master/Workers in Lua (1/2)

```
function Master(...)
  local nb_task, comp_size, comm_size, slave_count = unpack(arg)

  -- Dispatch the tasks
  for i = 1, nb_task do
    local tk = simgrid.task.new("Task " .. i, comp_size, comm_size)
    local alias = "worker " .. (i % worker_count)
    simgrid.info("Sending '" .. tk:get_name() .. "' to '" .. alias .. "'")
    tk:send(alias)
    simgrid.info("Done sending '" .. tk:get_name() .. "' to '" .. alias .. "'")
  end

  -- Sending finalize message to others
  for i = 0, worker_count - 1 do
    local alias = "worker " .. i;
    simgrid.info("Sending finalize to " .. alias)
    local finalize = simgrid.task.new("finalize", comp_size, comm_size)
    finalize:send(alias)
  end
end
```

Master/workers in Lua (2/2)

The worker

```
function Worker(...)
  local my_mailbox="worker " .. arg[1]

  while true do
    local tk = simgrid.task.recv(my_mailbox)
    if (tk:get_name() == "finalize") then
      simgrid.info("Got finalize message")
      break
    end
    tk:execute()
  end

  simgrid.info("Worker '" .. my_mailbox.. "': I'm done. See you!")
end
```

Setting up your experiment

```
require "simgrid"
simgrid.platform("my_platform.xml")
simgrid.application("my_deployment.xml")
simgrid.run()
simgrid.info("Simulation's over. See you.")
```

Master/Workers in Ruby (1/2)

Some mandatory headers

```
require 'simgrid'  
include MSG
```

The master

```
class Master < MSG::Process  
  def main(args)  
    numberOfTask = Integer(args[0])  
    taskComputeSize = Float(args[1])  
    taskCommunicationSize = Float(args[2])  
    workerCount = Integer(args[3])  
    for i in 0..numberOfTask-1  
      task = Task.new("Task_" + i.to_s, taskComputeSize , taskCommunicationSize);  
      mailbox = "worker " + (i%workerCount).to_s  
      MSG::info("Master Sending " + task.name + " to " + mailbox)  
      task.send(mailbox)  
      MSG::info("Master Done Sending " + task.name + " to " + mailbox)  
    end  
    for i in 0..workerCount-1  
      mailbox = "worker " + i.to_s  
      finalize_task = Task.new("finalize",0,0)  
      finalize_task.send(mailbox)  
    end  
  end  
end
```

Master/Workers in Ruby (2/2)

The worker

```
class Worker < MSG::Process
  def main(args)
    mailbox = "worker " + args[0]
    while true
      task = Task.receive(mailbox)
      if (task.name == "finalize")
        break
      end
      task.execute
      MSG::debug("Worker '" + mailbox + "' done executing task '" + task.name + ".")
    end
    MSG::info("I'm done, see you")
  end
end
```

Setting up your experiment

```
MSG.createEnvironment("platform.xml")
MSG.deployApplication("deploy.xml")
MSG.run
puts "Simulation time : " + MSG.getClock .to_s
MSG.exit
```

Some more polishing is needed

- ▶ Exceptions on timeout/host failure and so on?

Trace Replay: Separate your applicative workload

C code

```
static void action_blah(xbt_dynar_t parameters) { ... }
static void action_blih(xbt_dynar_t parameters) { ... }
static void action_bluh(xbt_dynar_t parameters) { ... }
int main(int argc, char *argv[]) {
    MSG_global_init(&argc, argv);
    MSG_create_environment(argv[1]);
    MSG_launch_application(argv[2]);
    /* No need to register functions as usual: actions started anyway */
    MSG_action_register("blah", blah);
    MSG_action_register("blih", blih);
    MSG_action_register("bluh", bluh);

    MSG_action_trace_run(argv[3]); // The trace file to run
}
```

Deployment

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/DTD/platform.dtd" [
<platform version="3">
  <process host="Tremblay" function="toto"/>
  <process host="Jupiter" function="tutu"/>
  <process host="Fafard" function="tata"/>
</platform>
```

Trace file

```
tutu blah toto 1e10
toto blih tutu
tutu bluh 12
toto blah 12
```

Trace Replay (2/2)

Separating the trace of each process

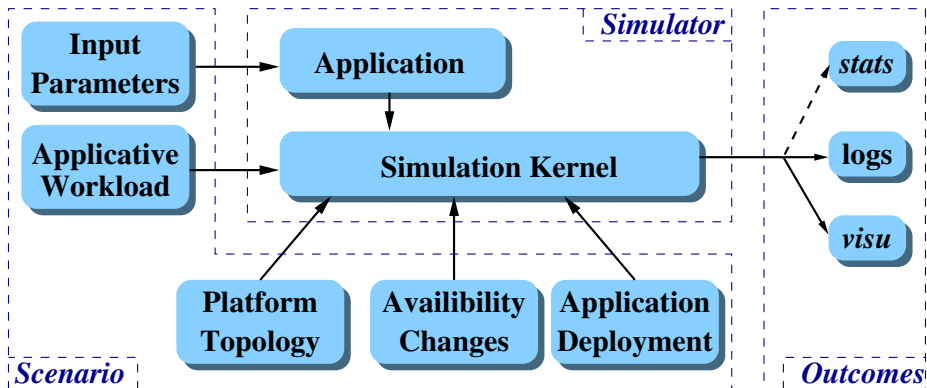
- ▶ Because it's sometimes more convenient (for MPI, you'd have to merge them)
- ▶ Simply pass NULL to `MSG_action_trace_run()`
- ▶ Pass the trace file to use as argument to each process in deployment

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
  <process host="Tremblay" function="toto">
    <argument value="actions_toto.txt"/>
  </process>
  <process host="Jupiter" function="tutu">
    <argument value="actions_tutu.txt"/>
  </process>
</platform>
```

Action Semantic

- ▶ This mechanism is completely agnostic: attach the meaning you want to events
- ▶ In `examples/actions/action.c`, we have pre-written event functions for:
 - ▶ **Basics:** send, recv, sleep, compute
 - ▶ **MPI-specific:** isend, irecv, wait, barrier, reduce, bcast, allReduce

SimGrid is not a Simulator



That's a Generic Simulation Framework

Configuring your simulators

Every simulator using SimGrid accepts a set of options

- `-help`: get some help
- `-help-models`: long help on models
- `-log`: configure the verbosity
- `-cfg`: change some settings

Note: SMPI-specific settings, are only visible in SMPI simulators

The log argument

- ▶ It's similar to Log4J, but in C
- ▶ You can increase the amount of output for some specific parts of SimGrid
- ▶ **Example:** See everything by using `-log=root.thres:debug`
- ▶ **List of all existing channels:** doc/html/group__XBT__log__cats.html

XBT from 10,000 feet

C is a basic language: we reinvented the wheel for you

Logging support: Log4C

```
XBT_LOG_NEW_DEFAULT_CATEGORY(test,
    "my own little channel");
XBT_LOG_NEW_SUBCATEGORY(details, test,
    "Another channel");

INFO1("Value: %d", variable);
CDEBUG3(details, "blah %d %f %d", x,y,z);
```

Exception support

```
xbt_ex_t e;
TRY {
    block
} CATCH(e) {
    block /* DO NOT RETURN FROM THERE */
}
```

Debugging your code

- ▶ Ctrl-C once: see processes' status
- ▶ Press it twice (in 5s): kill simulator

xbt_backtrace_display_current()

```
Backtrace (displayed in thread 0x90961c0):
---> In master() at masterslave_mailbox.c:35
---> In ?? ([0x4a69ba5])
```

Advanced data structures

- ▶ Hash tables (Perl's ones)
- ▶ Dynamic arrays, FIFOs
- ▶ SWAG (don't use); Graphs

String functions

- ▶ `bprintf`: `malloc()`ing `sprintf`
- ▶ `trim`, `split`, `subst`, `diff`
- ▶ string buffers

Threading support

- ▶ Portable wrappers (Lin, Win, Mac, Sim)
- ▶ Synchro (mutex, conds, semaphores)

Other

- ▶ Mallocators
- ▶ Configuration support
- ▶ Unit testing (check `src/testall`)
- ▶ Integration tests (`tesh`: testing shell)

Bindings Performance

What about performance loss for Java?

(Warning: these values are several years old)

workers \ 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. . .

What about the others?

- ▶ Very old preliminary results for Master/workers (10 workers; 200,000 tasks):

C (native)	7s
Lua	10.5s
Ruby	45s
Java	47s

- ▶ That's improvable
- ▶ It's garbage-collected
- ▶ User stack is dynamic in lua&ruby(?)
⇒ better scalability?

Conclusion: Finding the documentation

Conclusion: Finding the documentation

User manuals are for wimps

- ▶ Real Men read some slides 'cause they are more concise
- ▶ They read the examples, pick one modify it to fit their needs
- ▶ They may read 2 or 5% of the reference guide to check the syntax
- ▶ In doubt, they just check the source code

Conclusion: Finding the documentation

User manuals are for wimps

- ▶ Real Men read some slides 'cause they are more concise
- ▶ They read the examples, pick one modify it to fit their needs
- ▶ They may read 2 or 5% of the reference guide to check the syntax
- ▶ In doubt, they just check the source code

Users don't read the manual either

- ▶ **Proof:** that's why the RTFM expression were coined out
- ▶ Instead, they always ask same questions to lists, and get pointed to the FAQ

Conclusion: Finding the documentation

User manuals are for wimps

- ▶ Real Men read some slides 'cause they are more concise
- ▶ They read the examples, pick one modify it to fit their needs
- ▶ They may read 2 or 5% of the reference guide to check the syntax
- ▶ In doubt, they just check the source code

Users don't read the manual either

- ▶ **Proof:** that's why the RTFM expression were coined out
- ▶ Instead, they always ask same questions to lists, and get pointed to the FAQ

So, where is all SimGrid documentation?

- ▶ The SimGrid tutorial is a 200 slides presentation (motivation, models, example of use, internals)
- ▶ Almost all features of UAPI are demoed in an example (coverage testing)
- ▶ The reference guide contains a lot in introduction sections (about XBT)
- ▶ The FAQ contains a lot too; The code is LGPL anyway
- ▶ (actually, our documentation is not that bad. is it?)