batsim Documentation

Batsim team

Oct 16, 2021

Tutorials

1 Running your first simulation 3

2 Analyzing Batsim results 7

3 Implementing your scheduling algorithm 11

4 Modelling a stencil application in Batsim 13

5 Doing a reproducible experiment 17

6 Installation 29

7 Command-line Interface 35

8 Protocol 37

9 Progression of simulation time 57

10 Using Redis 63

11 File System model 65

12 Interval set 69

13 Frequenty Asked Questions (FAQ) 71

14 Changelog 75

15 Contact us 85

16 Platform 87

17 Workload 89

18 External Events 97

19 Schedule 99

20 Jobs 101

21 Energy 103

22 Contributing guidelines 105

23 Setting up a development environment 107

24 How to run Batsim tests? 115

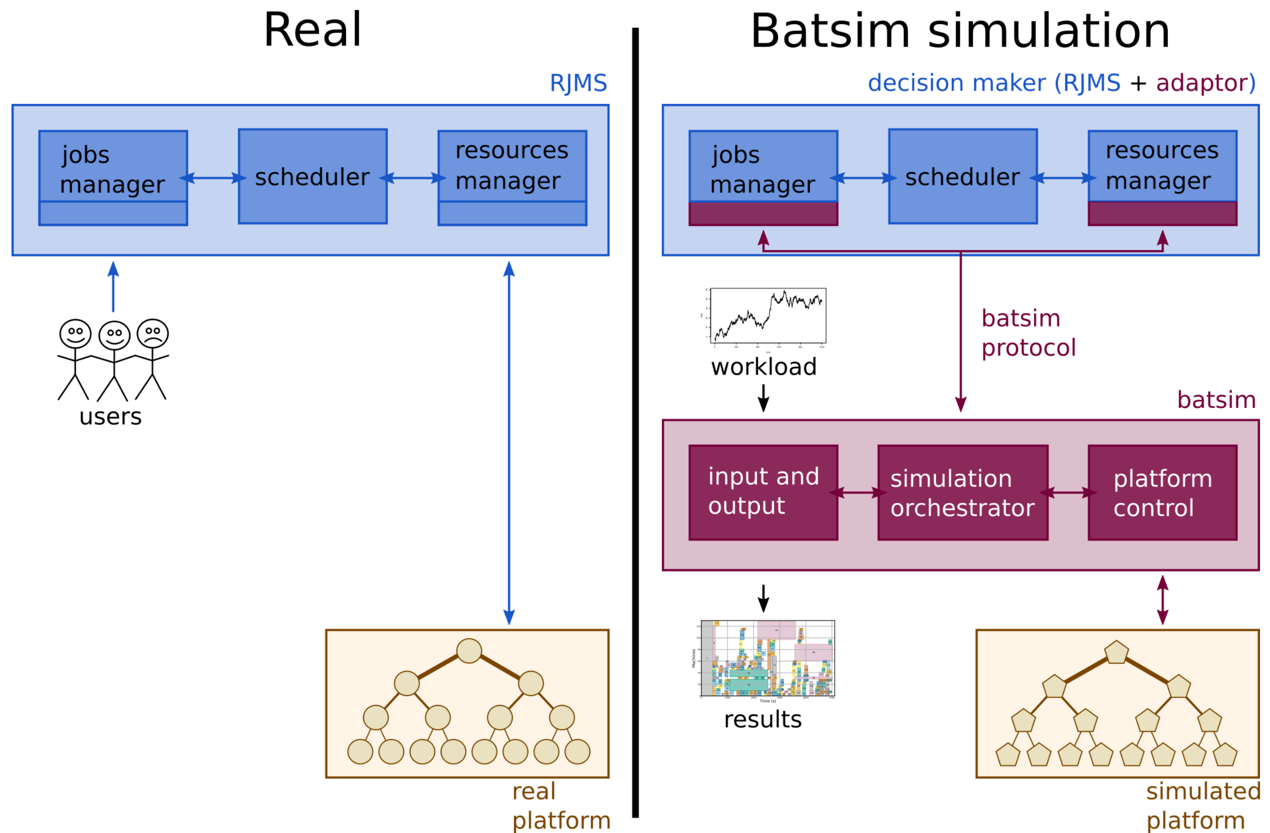
25 Continuous Integration 117

26 Release procedure 119

27 TODOs 121

Note: Parts of this documentation are still missing. Feel free to *Contact us* if you have any question or remark — or if a TODO is important for you. Here are *Contributing guidelines*.

Batsim is a scientific simulator to analyze batch schedulers. Batch schedulers — or Resource and Jobs Management Systems, RJMSs — are systems that manage resources in large-scale computing centers, notably by scheduling and placing jobs.



- Analyze and compare online scheduling algorithms.
- Sound simulation models thanks to SimGrid.
- Develop algorithms (in any programming language) without SimGrid knowledge, or to plug existing algorithm implementations to Batsim. Done thanks to a *Protocol* between Batsim and the schedulers
- Several ways to model how jobs should be simulated. Allows multiple levels of realism regarding several phenomena. Highly customizable to your needs.
- Keeping the implementation robust and maintainable is important to us.

The present documentation focuses on Batsim technical aspects. The most up-to-date scientific description of Batsim is done in Millian Poquet's PhD thesis (chapters 3 and 4). There is also the outdated Batsim initial white paper — please cite it (bibtex) if you use Batsim for your research.

CHAPTER 1

Running your first simulation

1.1 Prerequisites

This tutorial assumes that you have installed Batsim, batsched (a set of schedulers for Batsim) and robin (an experiment manager for Batsim).

The following Nix command installs the three aforementioned packages, but please read *Installation* first — as it may save you a lot of compilation time.

```
nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA batsim␣
↪batsched batexpe
```

1.2 Checking software environment

You can run the following commands to make sure that the different tools are installed at the expected version.

```
echo "batsim: $(batsim --version)"
echo "simgrid: $(batsim --simgrid-version)"
echo "batsched: $(batsched --version)"
echo "robin: $(robin --version)"
```

The version numbers should be the following.

```
batsim: 4.0.0
simgrid: 3.25.0
batsched: 1.4.0
robin: v1.2.0
```

1.3 Retrieving simulation inputs

Now that the different programs can be used, simulation inputs are needed to execute the simulation. This tutorial uses a simulated platform and a workload that are both defined in Batsim's repository. Retrieving these files is essentially a clone to Batsim's repository at latest release.

```
# Download a tarball of Batsim's latest release.
batversion='4.0.0'
curl --output "/tmp/batsim-v${batversion}.tar.gz" \
    "https://framagit.org/batsim/batsim/-/archive/v${batversion}/batsim-v${batversion}
↪.tar.gz"

# Extract tarball to /tmp/batsim-src-stable.
(cd /tmp && tar -xf "batsim-v${batversion}.tar.gz" && mv "batsim-v${batversion}"␣
↪batsim-src-stable)
```

1.4 Preparing the simulation output directory

This is simply the creation of the directory in which the simulation results will be put.

```
# Create the directory if needed.
mkdir -p /tmp/expe-out

# Clean the directory's content if any.
rm -rf /tmp/expe-out/*
```

1.5 Executing the simulation manually

A Batsim simulation most of the time involves two different processes.

1. Batsim itself, in charge of simulating what happens on the platform
2. A decision process, in charge of taking all the decisions about jobs and resources. This is where scheduling algorithms are implemented.

As running the simulation involves two processes, first launch two terminals. Batsim can be executed in the first terminal with the following command.

```
batsim -p /tmp/batsim-src-stable/platforms/cluster512.xml \
       -w /tmp/batsim-src-stable/workloads/test_batsim_paper_workload_seed1.json \
       -e "/tmp/expe-out/out"
```

The scheduler can now be run **in the second terminal** to start the simulation. In this tutorial we will use the batsched decision process, which is the reference implementation of a Batsim decision process. The following command runs batsched with the EASY backfilling algorithm.

```
batsched -v easy_bf
```

The simulation should now start. Both processes should finish soon. The simulation results should now be written in **EXPE_RESULT_DIR**.

1.6 Executing the simulation with robin

Executing the simulation manually shows how to run one Batsim simulation by executing the two processes manually. This is important to understand how running a simulation works, but in most cases executing them by hand is not desired.

robin has been created to make running Batsim simulation easy and robust. It allows to see Batsim simulations as black boxes, which is convenient when simulation campaigns are to be conducted.

One way to use robin is to first define how the simulation should be executed then to execute the simulation. Generating the **./expe.yaml** file, which describes the simulation instance description executed in *Executing the simulation manually*, can be done thanks to the following command.

```
robin generate ./expe.yaml \
      --output-dir=/tmp/expe-out \
      --batcmd="batsim -p /tmp/batsim-src-stable/platforms/cluster512.xml -w /tmp/
↪batsim-src-stable/workloads/test_batsim_paper_workload_seed1.json -e /tmp/expe-out/
↪out" \
      --schedcmd='batsched -v easy_bf'
```

Running the simulation can then be done with the following command.

```
robin ./expe.yaml
```

Robin's displayed output shows the lifecycle of Batsim and the scheduler. Robin returns 0 if and only if the simulation has been executed successfully.

Robin can directly execute a simulation without creating a description file, and have different options that makes it easy to call from experiment scripts. Please refer to `robin --help` for more information about these features.

CHAPTER 2

Analyzing Batsim results

2.1 Prerequisites

This tutorial assumes that you completed tutorial *Running your first simulation* and have kept the simulation results obtained during the tutorial.

2.2 Files overview

All Batsim output files are textual and are written in the same directory — they actually share their path prefix (see *Command-line Interface*).

- `PREFIX_jobs.csv` is the main output file. Contains information about the execution of each job (see *Jobs*).
- `PREFIX_schedule.csv` contains aggregated information about the whole simulation — such as makespan, mean waiting time or total consumed energy (see *Schedule*).
- `PREFIX_schedule.trace` is a Pajé trace of the simulation. Can be visualized with tools such as ViTE.
- `PREFIX_machine_states.csv` is a time series about the platform usage. It stores how many machines are in each state for each time interval. This file is mostly used to have a scalable view of the platform usage over time — this is useful when the number of jobs is big.

2.3 Computing some statistics

Most Batsim output files are plain CSV and can therefore be loaded in any data analysis framework.

The following script outlines how to do a basic analysis with in R without losing sanity thanks to tidyverse. The conclusions are of course not amazing on this toy workload.

```
#!/usr/bin/env Rscript
library('tidyverse') # Use the tidyverse library.
theme_set(theme_bw()) # Cosmetics.

jobs = read_csv('out_jobs.csv') # Read the jobs file.

# Manually compute some metrics on each job.
jobs = jobs %>% mutate(slowdown = (finish_time - starting_time) /
                                  (finish_time - submission_time),
                       longer_than_one_minute = execution_time > 60)

# Manually compute aggregated metrics.
# Here, the mean waiting time/slowdown for jobs with small execution time.
metrics = jobs %>% filter(longer_than_one_minute == FALSE) %>%
    summarize(mean_waiting_time = mean(waiting_time),
              mean_slowdown = mean(slowdown))

print(metrics) # Print aggregated metrics.

# Visualize what you want...
# Is there a link between jobs' waiting time and size?
ggplot(jobs) +
    geom_point(aes(y=waiting_time, x=requested_number_of_resources)) +
    ggsave('plot_wt_size.pdf')

# Is this still true depending on job execution time?
ggplot(jobs) +
    geom_point(aes(y=waiting_time, x=requested_number_of_resources)) +
    facet_wrap(~longer_than_one_minute) +
    ggsave('plot_wt_size_exectime.pdf')

# Is there a link with job size and execution time?
ggplot(jobs) +
    geom_violin(aes(factor(requested_number_of_resources), execution_time)) +
    ggsave('plot_exectime_size.pdf')
```

The script can be executed from the experiment output directory. It should print some metrics and generate several plots in the current directory.

Todo: We may think of more interesting things to plot while remaining simple. This is not easy on this toy workload though…

Maybe include the plot in this document if it is interesting.

2.4 Visualizing Gantt charts

Gantt charts can easily be visualized thanks to the Evalys Python library.

Evalys can take as input a regular SWF workload file or the Batsim `PREFIX_jobs.csv` output file to plot the Gantt chart of the jobs with the following script.

More detailed plots are presented in the examples of the Evalys repository.

```
from evalys.jobset import JobSet
from evalys import visu
js = JobSet.from_csv("PREFIX_jobs.csv")
visu.gantt.plot_gantt(js, label_jobs=True)
```

Todo: Introduce ViTE here and show an output example.

2.5 Build your own visualization

Todo: Talk about Evalys / custom scripts

CHAPTER 3

Implementing your scheduling algorithm

Studying the scheduling algorithm of your dreams in Batsim involves different things.

1. Implementing the scheduling algorithm itself.
2. Allowing the algorithm to communicate with Batsim through the *Protocol*.
3. Analyzing the algorithm behavior.

This tutorial focuses on the first two parts — as the third is done in *Analyzing Batsim results*.

3.1 Gaining insight on the Batsim protocol

The Batsim *Protocol* defines the interactions between Batsim and the scheduling algorithm. **Reading the protocol description is strongly recommended**, as it defines the following.

- The events the algorithm can react to.
- The actions that the algorithm can take.

3.2 Add your algorithm in a project already compatible with Batsim

This is the easiest way to implement your algorithm. Several projects gather collections of scheduling algorithms and implement the Batsim *Protocol* to communicate with the Batsim simulator.

Project	Language	Pros	Cons
batsched	C++	Kept **up-to-date** with most Batsim changes (used by Batsim tests).	C++: Error-prone, painful to write and maintain.
pybatsim	Python	Python: Great for **prototyping**.	Code lacks consistency. Python: Painful for robustness?
datsched	D	D: Great **(dev productivity, code maintenability)** tradeoff.	Uses **old protocol** (1.4.0). Just a prototype for now.
batsim_rust	Rust	Rust: **Safe and fast**.	Uses **old protocol** (1.4.0).

Todo: Hacking guidelines are not written yet about these projects.

Do not hesitate to *Contact us* about it.

3.3 Create a new project from scratch

This can be needed if you want to use another programming language, or it may be justified if the design decisions of existing implementations does not match what you want to do.

In this case, your best bet is to refer to the *Protocol* and to have a look at existing implementations.

CHAPTER 4

Modelling a stencil application in Batsim

This tutorial shows how a parallel application with simple patterns can be modelled in a Batsim workload. Here, we will assume that the target application is a simulation using Iterative Stencil Loops.

First, keep in mind that **there is no right way to model an application in Batsim**. How to model an application mostly depends on what phenomena you want to observe and at which granularity. Here is the context used in this tutorial.

- Stay coarse grain. We plan to use this application inside workloads to study scheduling algorithms, we do not plan to gain insight on the application itself.
- We want the application to be sensitive to its execution context: If any used resource is under congestion while the the application runs, its execution time should be longer.
- We want the application to impact how CPUs, network links, switches and disks are loaded over time. In other words, we want the application to potentially degrade the performance of other applcations and itself.

4.1 Details on the imaginary application

The imaginary application studied here follows an Iterative Stencil Loops pattern. The application takes a two-dimensional matrix of 32-bit floating-point values as input, of fixed size 4096×4096 elements. It computes 1000 *iterations* on the matrix. Each iteration consists in updating each value in the matrix according to the values of its neighbors. The application is parallel and always uses 4 processes. Spatial parallelism is used here: Each process works on a fourth of the total matrix. The sub-matrices held in memory by each process are not totally disjoint: Each process has a tiny border (1 row or column) of data coming from neighboring processes, which is exchanged between neighboring processes at each iteration.

Temporally speaking, the big steps of the applications are the following.

1. Spawn the 4 parallel processes. In real life, this would be done by `mpirun` or similar.
2. Each parallel process reads input data from a parallel file system. Here, all processes read the same amount of data. The total matrix size is $4096 \times 4096 \times 4 = 67108864$ bytes, and each process should read $2048 \times 2048 \times 4 = 16777216$ bytes.

3. All processes do a series of 100 iterations. Each iteration consists of the following.

1. Each process updates the values of the sub-matrix it is responsible for. Let us say this computation takes 10 floating-point operations for each value, which amounts for around 10^7 floating-point operations for the whole sub-matrix.

2. Each process shares the borders of its sub-matrix to its direct neighbors. Here, as we only have 4 processes, each process communicates with 2 other processes as shown on the figure above. The data transferred from one process to its neighbors is a row or column of the sub-matrix — *i.e.*, $2048 \times 4 = 8192$ bytes.

4. Every 100 iterations, a checkpoint of the current data is done. This is a common fault tolerance practice, so that if a machine fails the computation can be restarted from *quite recent* data, not the initial one. This checkpoint consists in writing the whole matrix on a parallel file system. The total amount of data transferred is therefore $4096 \times 4096 \times 4 = 67108864$ bytes, with $2048 \times 2048 \times 4 = 16777216$ bytes per process.

5. Once all 1000 iterations are done the application stops (the last checkpoint is the final application output).

4.2 Job modelling

A Batsim *Workload* consists of jobs and profiles.

- Jobs define the view of the application from the scheduler: This is a user request.
- Profiles define the view of the application from Batsim: This is how to simulate the application.

Here is a job description of such an application:

```
{"id":"0", "subtime":0, "walltime": 3600, "res": 4, "profile": "imaginary_stencil"}
```

- As said previously, this application is always run on `4` processes (the `res` key).
- We arbitrarily chose the job to be identified by `0` and to have a walltime of `3600` seconds — If it takes longer than this to be executed, Batsim will kill it automatically.
- Finally, we say that the job will be simulated using the `imaginary_stencil` profile, which is detailed in the next section.

4.3 Profile modelling

Let us model the various subparts of the application.

4.3.1 Data transfers with the parallel file system

First, the initial load of the matrix can be modelled with a *Homogeneous parallel tasks with IO to/from a Parallel File System* profile like this, which will read `67108864` bytes from a storage host called `pfs` and homogeneously split the data towards all the hosts involved in the job (here, `4` hosts).

```
{
  "type": "parallel_homogeneous_pfs",
  "bytes_to_read": 67108864,
  "bytes_to_write": 0,
  "storage": "pfs"
}
```

The checkpoints of the data produced by the application are very similar, the operation is just a `write` instead of a `read`.

```
{
  "type": "parallel_homogeneous_pfs",
  "bytes_to_read": 0,
  "bytes_to_write": 67108864,
  "storage": "pfs"
}
```

4.3.2 Iteration

The communication part of each iteration can be modelled with a *Parallel task* profile, which corresponds to the data transferred between each pair of hosts in the job (here, `4` hosts). The data here corresponds to the transfer of 1 row/column between neighboring processes, as seen on the application details figure.

```
{
  "type": "parallel",
  "cpu": [    0,    0,    0,    0],
  "com": [    0, 8192, 8192,    0,
           8192,    0,    0, 8192,
           8192,    0,    0, 8192,
              0, 8192, 8192,    0]
}
```

The computation part of each iteration can also be modelled with a *Parallel task* profile such as the following.

```
{
  "type": "parallel",
  "cpu": [ 1e7,  1e7,  1e7,  1e7],
  "com": [    0,    0,    0,    0,
              0,    0,    0,    0,
              0,    0,    0,    0,
              0,    0,    0,    0]
}
```

4.3.3 Putting all pieces together

Finally, we can see how to put the previous sub-profiles together to define the `imaginary_stencil` profile. There are many ways to do so. **In a real-life scenario on a real application, you should evaluate the prediction precision of the different choices** to decide which model fits your application (probably with a simulation overhead trade-off in mind). Here is a set of profiles that define `imaginary_stencil`, bundling a sequence of 100 iterations together in the same *Parallel task*.

```
{
  "initial_load": {
    "type": "parallel_homogeneous_pfs",
    "bytes_to_read": 67108864,
    "bytes_to_write": 0,
    "storage": "pfs"
  },
  "100_iterations": {
```

(continues on next page)

(continued from previous page)

```
    "type": "parallel",
    "cpu": [   1e9,     1e9,     1e9,     1e9],
    "com": [     0, 819200, 819200,       0,
          819200,       0,       0, 819200,
          819200,       0,       0, 819200,
               0, 819200, 819200,       0]
  },
  "checkpoint": {
    "type": "parallel_homogeneous_pfs",
    "bytes_to_read": 0,
    "bytes_to_write": 67108864,
    "storage": "pfs"
  },
  "iterations_and_checkpoints": {
    "type": "composed",
    "repeat": 10,
    "seq": ["100_iterations", "checkpoint"]
  },
  "imaginary_stencil": {
    "type": "composed",
    "repeat": 1,
    "seq": ["initial_load", "iterations_and_checkpoints"]
  }
}
```

- The `initial_load` and `checkpoint` profiles are exactly the same as in the *Data transfers with the parallel file system* section.
- The `100_iterations` *Parallel task* profile is the merge of 100 consecutive iterations into a single *Parallel task*.
- The `iterations_and_checkpoints` *Sequence of profiles* represents most of the application, as it repeats `10` times the execution of (`100_iterations` followed by a `checkpoint`).
- Finally, the `imaginary_stencil` profile represents the whole application, which is `initial_load` followed by `iterations_and_checkpoints`.

4.3.4 Impact of this modelling choice

Here, we chose to group series of 100 iterations together into a single *Parallel task*. This means that during the whole `100_iterations` *Parallel task*, the resource consumption induced by the profile is quite homogeneous: Communications and computations will be done at the same time, at the rate of the slowest of the two depending on the execution context. This means bursty phenomena induced by each iteration will not be observable and that the latency of communication phases will only be paid once, not once per iteration. However, as the `checkpoint` is run alone, the burst on the parallel file system induced by the checkpoints will be observable, as well as the latency to communicate with the parallel file system for each checkpoint.

CHAPTER 5

Doing a reproducible experiment

This tutorial shows how to execute a Batsim simulation in a fully reproducible software environment thanks to the Nix package manager. Two schedulers are used here, to show how to use a specific version of batsched and pybatsim.

5.1 Prerequisites

As a small Nix knowledge is recommended, we strongly encourage readers to follow a tutorial about Nix and reproducible experiments first. This tutorial introduces main Nix concepts and focuses on those useful for reproducible experiments, lowering the entry cost in the Nix ecosystem.

> **Warning:** This tutorial uses old versions of Batsim and scheduler. It should still work (otherwise this is a bug, please *Contact us* to report the issue) and the presented steps to improve the experiment repeatability are still relevant.
>
> However, please refer to *Using Batsim from a well-defined Nix environment* if you are looking for a starting environment to copy/paste for your experiment, **not** to these files.

5.2 Defining a reproducible environment

5.2.1 A first attempt

A direct attempt to define such an environment is to use the packages defined in NUR-Kapack. This is what the following Nix file describes.

Listing 1: `env-first-attempt.nix`

```
{ kapack ? import
    ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/master.tar.gz")
```

(continues on next page)

(continued from previous page)

```
  {}
}:

with kapack.pkgs;

let
  self = rec {
    experiment_env = mkShell rec {
      name = "experiment_env";
      buildInputs = [
        # simulator
        kapack.batsim
        # scheduler implementations
        kapack.batsched
        kapack.pybatsim
        # misc. tools to execute instances
        kapack.batexpe
      ];
    };
  };
in
  self.experiment_env
```

Assuming that the `env-first-attempt.nix` file is in your working directory, you can enter into the environment with the following command: `nix-shell --pure ./env-first-attempt.nix`. You should be able to call `batsim`, `batsched`, `pybatsim` and `robin` from inside this shell. The shell can be exited as usual (Ctrl+D, `exit`…).

While this shell perfectly works to use the latest release of the desired packages, it does not describe the desired versions well enough to be reproductible: If you use the same environment in the future, the versions may have changed.

5.2.2 Pinning the package repository

A first step to improve our setup is to use a specific *pinned* version of NUR-Kapack (the repository that contains package definitions) instead of its `master` branch.

Listing 2: `env-pin-pkgs-repo.nix`

```
{ kapack ? import
    ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/
↪1672831224a21d6c34350d8f78cff9266e3e28a2.tar.gz")
  {}
}:

with kapack.pkgs;

let
  self = rec {
    experiment_env = mkShell rec {
      name = "experiment_env";
      buildInputs = [
        # simulator
        kapack.batsim
        # scheduler implementations
```

(continues on next page)

(continued from previous page)

```
        kapack.batsched
        kapack.pybatsim
        # misc. tools to execute instances
        kapack.batexpe
      ];
    };
  };
in
  self.experiment_env
```

The `env-pin-pkgs-repo.nix` file is exactly the same as the previous one but about its `kapack` import definition: Kapack's commit 1672831 is used instead of the `master` branch. This makes sure that the versions of the desired packages (`batsim`, `batsched...`) will not change.

> **Warning:** This is not true if you use the `-master` variant of a package!
>
> For example, `batsim-master` represents Batsim built from Batsim's master branch's latest commit – without any kind of commit pinning.

5.2.3 Pinning the packages

In many cases you might want to use a specific version of a package rather than its latest release. Nix makes it easy by *overridding* a package definition, as in the following file.

Listing 3: `env-pin-everything.nix`

```
{ kapack ? import
    ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/
↪1672831224a21d6c34350d8f78cff9266e3e28a2.tar.gz")
  {}
}:

with kapack.pkgs;

let
  self = rec {
    my_batsim = kapack.batsim-310.overrideAttrs (attr: rec {
      name = "batsim-3.1.0-346e0de";
      src = fetchgit {
        url = "https://framagit.org/batsim/batsim.git";
        rev = "346e0de311c10270d9846d8ea418096afff32305";
        sha256 = "0jacrinzzx6nxm99789xjbip0cn3zfsg874zaazbmicbpllxzh62";
      };
    });
    my_batsched = kapack.batsched-130.overrideAttrs (attr: rec {
      name = "batsched-1.3.0-db0450a";
      src = fetchgit {
        url = "https://framagit.org/batsim/batsched.git";
        rev = "db0450a608656f0661f4c8d6f132c68b2d402a59";
        sha256 = "05bn43xrk4qz8w2v58zkl17vmj4y57y93lrgrpv7jsdkird9a5vw";
      };
    });
    my_pybatsim = kapack.pybatsim-320.overrideAttrs (attr: rec {
      name = "pybatsim-3.1.0-ca81c4a";
```

(continues on next page)

(continued from previous page)

```
      src = fetchgit {
        url = "https://gitlab.inria.fr/batsim/pybatsim.git";
        rev = "ca81c4a49b84fb5249367ae64bdc9289d619a033";
        sha256 = "153wxqyz2pgb3skspz9628s91zrsvbzvgxpx6c6sbjharavdnyik";
      };
    });

    experiment_env = mkShell rec {
      name = "experiment_env";
      buildInputs = [
        # simulator
        my_batsim
        # scheduler implementations
        my_batsched
        my_pybatsim
        # misc. tools to execute instances
        kapack.batexpe
      ];
    };
  };
in
  self.experiment_env
```

The `env-pin-everything.nix` file defines custom versions of Batsim, batsched and pybatsim — and uses them in the `experiment_env` shell. This is especially useful for using your own variant of a scheduling algorithm, as you can put the git repository of your choice in the `url` field of the `fetchgit` command and thus use your own fork of a scheduler project.

Note: Filling correctly the `sha256` field of the package source overriding can be annoying.

A fast way to find the right value is to first fill it with a random sha256 value (`echo hello | sha256sum`), then to try to enter your shell. Nix will whine about the hash mismatch then print the hash value it computed :). In the example below, Nix computed a sha256 value of `0jacrinzzx6nxm99789xjbip0cn3zfsg874zaazbmicbpllxzh62`.

```
hash mismatch in fixed-output derivation '/nix/store/vmqrfa7l1hg1lshaj15jz46li5v8r2qs-
↪batsim-346e0de':
  wanted: sha256:00xyyr3fi8l6hb839bv3f7yb86yjv7xi1cgh1xnhipym4asvb4aq
  got:    sha256:0jacrinzzx6nxm99789xjbip0cn3zfsg874zaazbmicbpllxzh62
```

5.3 Setting up a full experiment

Note: A rendering of this experiment notebook is hosted there.

This experiment example shows how Nix can be help in designing an experiment with a reproducible software environment. Its goal is to evaluate whether the introduction of smart pointers reduced Batsim's memory usage over time or not.

5.3.1 Environments

Listing 4: `env-check-memuse-improvement.nix`

```
#
# WARNING: do NOT use old kapack for new work!
# make sure you use NUR-kapack instead!
#
{ old_kapack ? import
  ( fetchTarball "https://github.com/oar-team/kapack/archive/
↪773d3909d78f1043ffb589a725773699210d71d5.tar.gz") {}
, kapack ? import
  ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/
↪1672831224a21d6c34350d8f78cff9266e3e28a2.tar.gz") {}
}:

with kapack.pkgs;

let
  self = rec {
    # an old version of Batsim, before the introduction of smart pointers.
    old_batsim = old_kapack.batsim_dev.overrideAttrs (attr: rec {
      name = "batsim-3.1.0-346e0de";
      src = fetchgit {
        url = "https://framagit.org/batsim/batsim.git";
        rev = "346e0de311c10270d9846d8ea418096afff32305";
        sha256 = "0jacrinzzx6nxm99789xjbip0cn3zfsg874zaazbmicbpllxzh62";
      };
      mesonBuildType = "release";
      hardeningDisable = [];
      dontStrip = false;
    });
    # a more recent Batsim, after the introduction of smart pointers.
    batsim = (kapack.batsim-310.override{simgrid=kapack.simgrid-325;}).overrideAttrs␣
↪(attr: rec {
      name = "batsim-3.1.0-5906dbe";
      src = fetchgit {
        url = "https://framagit.org/batsim/batsim.git";
        rev = "5906dbe67ba5c6229029e3ddcde5979ae116f287";
        sha256 = "08jwsgiz0s9n15pcv637sq31gyd3qzja850ycaz06kv59jlzcrfb";
      };
      mesonBuildType = "release";
      hardeningDisable = [];
      dontStrip = false;
    });
    # set of scheduling algorithms
    batsched = kapack.batsched-130.overrideAttrs (attr: rec {
      name = "batsched-1.3.0-dev";
      src = fetchgit {
        url = "https://framagit.org/batsim/batsched.git";
        rev = "54b18eb4f24bdb69617baa58b5b07842c70df094";
        sha256 = "08mw03k18m4ppschrcmyali33im4hz8060j990aa673vpdlf0pb2";
      };
    });

    # r tools around Batsim
    battools_r = kapack.pkgs.rPackages.buildRPackage {
      name = "battools-r-fcccf8a";
      src = fetchgit {
```

(continues on next page)

(continued from previous page)

```
        url = "https://framagit.org/batsim/battools.git";
        rev = "fcccf8a6bccae388af6a17b866bba6c11097734f";
        sha256 = "05vll6rhdiyg38in8yl0nc1353fz2j7vqpax64czbzzhwm5d5kfs";
      };
      propagatedBuildInputs = with kapack.pkgs.rPackages; [
        dplyr
        readr
        magrittr
        assertthat
      ];
    };

    # a python tool to transform massif traces to exploitable data
    massif_to_csv = kapack.pkgs.python3Packages.buildPythonPackage rec {
      pname = "massif_to_csv";
      version = "0.1.0";
      propagatedBuildInputs = [msparser];
      src = builtins.fetchurl {
        url = "https://files.pythonhosted.org/packages/09/2d/
↪674c3405939f198e963ba5e73c2a331ef3364bc52da9b123c8f16dd60c8d/massif_to_csv-0.1.0.
↪tar.gz";
        sha256 = "f5eb01dce6d2e4a6c9812fd58f0add20b6739ba340482b7902f311298eb37dfb";
      };
    };

    # a massif_to_csv dependency
    msparser = kapack.pkgs.python3Packages.buildPythonPackage rec {
      pname = "msparser";
      version = "1.4";
      buildInputs = [
        kapack.pkgs.python3Packages.pytest
      ];
      src = builtins.fetchurl {
        url = "https://files.pythonhosted.org/packages/e0/68/
↪aece1c5e75b49d95f304d2df029ae69583ef59a55694ec683e2452d70637/msparser-1.4.tar.gz";
        sha256 = "1199d27bdc492647d2d17d7776e49176f3ec3d2d959d4cfc8b2ce9257cefc16f";
      };
    };

    # dependencies in common for the two experimental environments
    common_expe_deps = [
      kapack.batexpe
      batsched
      valgrind
    ];

    # environment used to generate simulation inputs.
    input_preparation_env = mkShell rec {
      name = "input-preparation-env";
      buildInputs = [
        # to generate robin instances
        kapack.batexpe
        # to generate a batsim workload
        kapack.pkgs.R
        battools_r
        # to download a batsim platform
        curl
```

(continues on next page)

(continued from previous page)

```
      ];
    };
    # environment used to execute simulations with the old Batsim version.
    simulation_old_env = mkShell rec {
      name = "old-env";
      buildInputs = [old_batsim] ++ common_expe_deps;
    };
    # environment used to execute simulations with the recent Batsim version.
    simulation_env = mkShell rec {
      name = "env";
      buildInputs = [batsim] ++ common_expe_deps;
    };
    # environment used to analyze the results, and to render them into a html␣
↪document.
    notebook_env = mkShell rec {
      name = "notebook-env";
      buildInputs = [
        # Tools to analyse results.
        kapack.pkgs.R
        kapack.pkgs.rPackages.tidyverse
        kapack.pkgs.rPackages.viridis
        massif_to_csv
        # Rmarkdown-related tools (Rmarkdown is a notebook technology).
        kapack.pkgs.rPackages.knitr
        kapack.pkgs.rPackages.rmarkdown
        kapack.pkgs.pandoc
      ];
    };
  };
in
  self
```

The `env-check-memuse-improvement.nix` file describes various environments — comments describe the role of each environment. Several simulation environments are used here as we want to evaluate several versions of the same software (batsim), as using two versions of the same package in the same environment would create a collision.

5.3.2 Scripts

Similarly to what would (should) be done in a real experiment, some scripts are used here so that some steps are kept independent from each other and from the main engine used to run the experiment. Here, a rmarkdown notebook is used as the main engine to execute the simulation, and to analyze and present the results.

The scripts used here have the following content.

Listing 5: `generate-workload.R`

```
#!/usr/bin/env Rscript
library(battools)
library(dplyr)

# Read a SWF workload.
kth_sp2 = read_swf("http://www.cs.huji.ac.il/labs/parallel/workload/l_kth_sp2/KTH-SP2-
↪1996-2.1-cln.swf.gz")
```

(continues on next page)

(continued from previous page)

```
# Only work on a short period (a month) from an arbitrary time point.
date_begin = mean(kth_sp2$submit_time)
date_end = date_begin + 60*60*24*30*3
month = kth_sp2 %>% filter(submit_time >= date_begin) %>% filter(submit_time < date_
↪end)

# Generate a Batsim workload.
workload = swf_to_batworkload_delay(month, 100, subtime_strat = "translate_to_zero")
write_batworkload(workload, "./kth_month.json")
```

Listing 6: prepare-instances.bash

```
#!/usr/bin/env bash
set -eu

# start from a clean directory structure
EXPE_DIR=$(realpath ./expe)
rm -rf ${EXPE_DIR}

# create instances' directories
mkdir -p ${EXPE_DIR}/old
mkdir -p ${EXPE_DIR}/new

# generate a robin file for each instance
BATCMD_BASE="batsim -p '${EXPE_DIR}/cluster.xml' -w '${EXPE_DIR}/kth_month.json' --
↪mmax-workload"
robin generate "${EXPE_DIR}/old.yaml" \
      --output-dir "${EXPE_DIR}/old" \
      --batcmd "valgrind --tool=massif --time-unit=ms --massif-out-file='${EXPE_DIR}/
↪old/massif.out' ${BATCMD_BASE} -e '${EXPE_DIR}/old/out'" \
      --schedcmd "batsched -v easy_bf_fast"

robin generate "${EXPE_DIR}/new.yaml" \
      --output-dir "${EXPE_DIR}/new" \
      --batcmd "valgrind --tool=massif --time-unit=ms --massif-out-file='${EXPE_DIR}/
↪new/massif.out' ${BATCMD_BASE} -e '${EXPE_DIR}/new/out'" \
      --schedcmd "batsched -v easy_bf_fast"
```

Listing 7: run-notebook.R

```
#!/usr/bin/env Rscript
rmarkdown::render('notebook.Rmd')
```

5.3.3 Notebook

Finally, here is the notebook source:

Listing 8: notebook.Rmd

```
Batsim: Impact of smart pointers on Batsim's memory usage
=========================================================

This notebook is an example of a repeatable experiment from [one of Batsim's␣
↪documentation tutorial](https://batsim.readthedocs.io/en/latest/tuto-reproducible-
↪experiment/tuto.html).
```

(continues on next page)

(continued from previous page)

```
Simulation instances preparation
--------------------------------

Here, we want to run two simulations with the same inputs but with a different Batsim␣
↪version.
This can be done by executing the `prepare-instances.bash` script in its dedicated␣
↪environment:

```{bash}
nix-shell env-check-memuse-improvement.nix -A input_preparation_env --command './
↪prepare-instances.bash'
```

This creates the following files:
```{bash}
tree ./expe
```

Getting simulation inputs

Here we will simulate the old [KTH SP2 workload](https://www.cse.huji.ac.il/labs/
↪parallel/workload/l_kth_sp2/index.html) from the parallel workloads archive.
The `generate-workload.R` script downloads the raw logs, extracts a month in the␣
↪middle of the trace then generate a batsim workload from it. It is called in the␣
↪input preparation environment:

```{bash, results="hide"}
nix-shell env-check-memuse-improvement.nix -A input_preparation_env --command '(cd ./
↪expe && ../generate-workload.R)'
```

We will use a platform with enough resources from the Batsim repository.
Platform caracteristics do not matter much here, as we use delay profiles that are␣
↪not sensitive to the jobs execution context.

```{bash}
nix-shell env-check-memuse-improvement.nix -A input_preparation_env --command 'curl -
↪k -o ./expe/cluster.xml https://framagit.org/batsim/batsim/raw/
↪346e0de311c10270d9846d8ea418096afff32305/platforms/cluster512.xml'
```

Running simulations

This is done by executing robin on the instance files in their dedicated environments.
Please note that separating these two environments is mandatory, as a different␣
↪Batsim version is defined in each environment.

```{bash}
nix-shell env-check-memuse-improvement.nix -A simulation_env --command 'robin ./expe/
↪new.yaml'
nix-shell env-check-memuse-improvement.nix -A simulation_old_env --command 'robin ./
↪expe/old.yaml'
```
```

(continues on next page)

(continued from previous page)

```
Analyzing results
-----------------

First, we can visually check that the simulation results are similar.

```{r, message=FALSE, fig.width=10, fig.height=6}
library(tidyverse)
library(viridis)
theme_set(theme_bw())

# batsim-generated summaries
old_schedule = read_csv('./expe/old/out_schedule.csv') %>% mutate(instance='old')
new_schedule = read_csv('./expe/new/out_schedule.csv') %>% mutate(instance='new')
schedules = bind_rows(old_schedule, new_schedule)
schedules %>% tbl_df %>% rmarkdown::paged_table()

# jobs data
old_jobs = read_csv('./expe/old/out_jobs.csv') %>% mutate(instance='old')
new_jobs = read_csv('./expe/new/out_jobs.csv') %>% mutate(instance='new')
jobs = bind_rows(old_jobs, new_jobs) %>% mutate(color_id=job_id%%5)

jobs_plottable = jobs %>%
    mutate(starting_time = starting_time / (60*60*24),
           finish_time = finish_time / (60*60*24)) %>%
    separate_rows(allocated_resources, sep=" ") %>%
    separate(allocated_resources, into = c("psetmin", "psetmax"), fill="right") %>%
    mutate(psetmax = as.integer(psetmax), psetmin = as.integer(psetmin)) %>%
    mutate(psetmax = ifelse(is.na(psetmax), psetmin, psetmax))

jobs_plottable %>%
    ggplot(aes(xmin=starting_time,
               ymin=psetmin,
               ymax=psetmax + 0.9,
               xmax=finish_time,
               fill=color_id)) +
    geom_rect(alpha=0.9, color="black", size=0.1, show.legend = FALSE) +
    scale_fill_viridis() +
    facet_wrap(~instance, ncol=1) +
    labs(x='Simulation time (day)', y="Resources") +
    ggsave('./gantts.pdf', width=15, height=9)
```

Aggregated metrics are the same and the Gantt charts look similar.

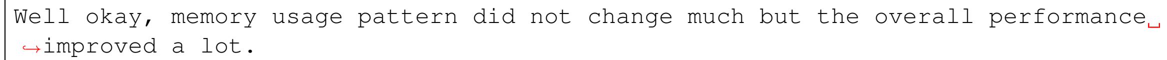
Let us now give a look at Batsim's memory footprint over time for both runs.
```{bash}
massif-to-csv ./expe/old/massif.out{,.csv}
massif-to-csv ./expe/new/massif.out{,.csv}
```

```{r, message=FALSE, fig.width=10, fig.height=6}
old_massif = read_csv('./expe/old/massif.out.csv') %>% mutate(instance='old')
new_massif = read_csv('./expe/new/massif.out.csv') %>% mutate(instance='new')
massif = bind_rows(old_massif, new_massif) %>% mutate(
    total=(stack+heap+heap_extra) / 1e6,
    time=time/1e3)
```

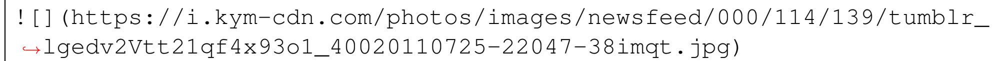
(continues on next page)
```

(continued from previous page)

```
massif %>%
    ggplot(aes(x=time, y=total)) +
    geom_step() +
    facet_wrap(~instance, ncol=1) +
    labs(x='Real time (s)', y="Batsim process's memory consumption (Mo)") +
    ggsave('./memuse_over_time.png', width=15, height=9)
```



```
Well okay, memory usage pattern did not change much but the overall performance
↪improved a lot.
```



```
![](https://i.kym-cdn.com/photos/images/newsfeed/000/114/139/tumblr_
↪lgedv2Vtt21qf4x93o1_40020110725-22047-38imqt.jpg)
```

The notebook can be run with the following command — which will run the simulations and analyses from the notebook.

```
nix-shell env-check-memuse-improvement.nix -A notebook_env --command ./run-notebook.R
```

CHAPTER 6

Installation

This page presents how to install Batsim and some of its tools. We recommend *Using Nix*, but you may prefer *Using Batsim from a Docker container*, *Using Batsim with Singularity* or just *Building it yourself*.

6.1 Using Nix

Batsim and its ecosystem are packaged in the kapack repository. These packages use the Nix package manager. We recommend to use Nix as its purity property allows to fully define all the software dependencies of our tools — as well as the versions of each software. This property is great to produce controlled software environments, as showcased in *Doing a reproducible experiment*.

If you already have a working Nix installation, you can skip *Installing Nix* and directly go for *Using Batsim from a well-defined Nix environment* or *Installing Batsim in your system via Nix*.

Note: Most package have at least two versions in kapack, named `PACKAGE` and `PACKAGE-master`. `PACKAGE` stands for the latest release of the package, while the `-master` version is the latest unstable commit from the main git branch.

Note: Nix commands check if the requested packages — **and ALL their dependencies** — are already available in your system. If this is not the case, all the required packages will be built, which may take a lot of time. To speed the process up you can enable the use of Batsim's binary cache, so that Nix will download our binaries instead of rebuilding them.

```
# Install cachix (using Nix).
# (up-to-date instructions are there: https://cachix.org/)
nix-env -iA cachix -f https://cachix.org/api/v1/install

# Configure cachix: tell Nix to use batsim.cachix.org as a binary cache.
cachix use batsim
```

6.1.1 Installing Nix

Note: This is unlikely but the procedure to install Nix may be outdated. Please refer to Nix installation documentation for up-to-date installation material.

Installing Nix is pretty straightforward.

```
curl -L https://nixos.org/nix/install | sh
```

Follow the instructions displayed at the end of the script. You usually need to `source` a file to access the Nix commands.

Warning: On some distributions like Debian 10, kernel user namespaces are disabled. **They should be enabled to make sure Nix works properly.** Enable them with the following command:

```
sudo echo 1 > /proc/sys/kernel/unprivileged_userns_clone
```

6.1.2 Using Batsim from a well-defined Nix environment

Defining a software environment from which your simulations run is quite straightforward with Nix. **This is the recommended way to use Batsim.**

For example, the following file defines an en environment from which you can execute batsim and a scheduler implementation. It uses the last release of our tools.

Listing 1: `tuto-env.nix`

```
let
  kapack = import
    ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/master.tar.gz") {};
in

kapack.pkgs.mkShell rec {
  name = "tuto-env";
  buildInputs = [
    kapack.batsim # simulator
    kapack.batsched # scheduler
    kapack.batexpe # experiment management tools
  ];
}
```

After downloading an environment file on your machine's filesystem, you can enter the environment thanks to `nix-shell` — for example `nix-shell ./env.nix` if you downloaded `tuto-env.nix` into you current working directory as `env.nix`. The strength of this approach is that you can easily tune which version you want to use for each tool. For example, the next environment uses the latest commit of the master branch of the same tools.

Listing 2: `tuto-env-master.nix`

```
let
  kapack = import
    ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/master.tar.gz") {};
in
```

(continues on next page)

(continued from previous page)

```
kapack.pkgs.mkShell rec {
  name = "tuto-env-master";
  buildInputs = [
    kapack.batsim-master # simulator
    kapack.batsched-master # scheduler
    kapack.batexpe-master # experiment management tools
  ];
}
```

And here is a more advanced example where you can specify the git repository and commit to use for every tool. This can be used as a base to start an experiment using Batsim, as you will probably need to implement new algorithms/features in a scheduler, Batsim or both. You may also be interested in reading *Doing a reproducible experiment* for good practice advices.

Listing 3: `tuto-env-pinned.nix`

```
let
  kapack = import
    ( fetchTarball "https://github.com/oar-team/nur-kapack/archive/master.tar.gz") {};
in

let my-packages = rec {
  # define your own batsim version from batsim-4.0.0
  # (you can select another base, such as batsim-310 for batsim-3.1.0)
  my-batsim = kapack.batsim-400.overrideAttrs(old: rec {
    # define where to the source code of your own batsim version.
    # here, use a given commit on the official batsim git repository on framagit,
    # but you can of course use your own commit/fork instead.
    version = "c79d95851cd8f80089b7218a2cf85f4b26ebde5f";
    src = kapack.pkgs.fetchgit rec {
      url = "https://framagit.org/batsim/batsim.git";
      rev = version;
      sha256 = "0lr3vawdbmajisgvbj5cjqw0wfy9y990yhl20kplarambx40nplp";
    };
  });

  # define your own version of a scheduler, here from batsched-1.4.0
  my-batsched = kapack.batsched-140.overrideAttrs(old: rec {
    # (you can also your own commit/fork of batsched here)
    version = "b020e48b89a675ae681eb5bcead01035405b571e";
    src = kapack.pkgs.fetchgit rec {
      url = "https://framagit.org/batsim/batsched.git";
      rev = version;
      sha256 = "1dmx2zk25y24z3m92bsfndvvgmdg4wy2iildjmwr3drmw15s75q0";
    };
  });

  # can also be done on the pybatsim scheduler, here from pybatsim-3.2.0
  # (note that overridePythonAttrs is needed instead of overrideAttrs)
  my-pybatsim = kapack.pybatsim-320.overridePythonAttrs(old: rec {
    # (you can also your own commit/fork of pybatsim here)
    version = "fa6600eccd4e5ffbebfc7ecba05b64cf84c5e9d3";
    src = kapack.pkgs.fetchgit rec {
      url = "https://gitlab.inria.fr/batsim/pybatsim.git";
      rev = version;
```

(continues on next page)

(continued from previous page)

```
      sha256 = "0nr8hqsx6y1k0lv82f4x3fjq6pz2fsd3h44cvnm7w21g4v3s6l6y";
    };
  });

  shell = kapack.pkgs.mkShell rec {
    name = "tuto-env";
    buildInputs = [
      my-batsim
      my-batsched
      my-pybatsim
      kapack.batexpe
    ];
  };
};
in
  my-packages.shell
```

6.1.3 Installing Batsim in your system via Nix

This can be done with `nix-env --install`.

```
# Install the Batsim simulator.
nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA batsim

# Other packages from the Batsim ecosystem can also be installed this way.
# For example schedulers.
nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA batsched
nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA pybatsim

# Or interactive visualization tools.
nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA evalys

# Or experiment management tools...
nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA batexpe
```

6.2 Using Batsim from a Docker container

Batsim and all its runtime dependencies are packaged in the oarteam/batsim Docker container, which allows to run batsim without any installation on a Linux host — assuming that Docker is installed.

```
docker run \
    --net host \
    -u $(id -u):$(id -g) \
    -v $PWD:/data \
    oarteam/batsim:latest \
    -p /data/platf.xml -w /data/wload.json
```

Here is a quick explanation on the various parameters.

- `--net host` enables the use of the host network to communicate with the scheduler.
- `-u $(id -u):$(id -g)` enables the generation of output files with your own user permission.
- `-v $PWD:/data` shares your local directory so batsim can find input files and write output files.

- `oarteam/batsim:latest` is the image to run. `latest` is built from master branch's last commit.
- `-p /data/platf.xml -w /data/wload.json` are batsim arguments (see *Command-line Interface*).

6.3 Using Batsim with Singularity

The oarteam/batsim docker containers can directly be executed by Singularity.

```
singularity exec \
    docker://oarteam/batsim:latest \
    batsim -p /data/platf.xml -w /data/wload.json
```

6.4 Building it yourself

Batsim can be built with the Meson (+ Ninja) build system. You can also use CMake if you prefer but please note that our cmake support is deprecated.

Warning: You first need to install all Batsim dependencies for the following lines to work.

- Decent clang/gcc (real C++17 support).
- Decent boost.
- Recent SimGrid.
- ZeroMQ.
- Redox (our fork has pkg-config support) and its dependencies (hiredis, libev).
- RapidJSON.
- Pugixml.
- Docopt.cpp.

Make sure you install versions of these packages with pkg-config support! The two build systems we use rely on pkg-config to find dependencies.

The dependency list above may be outdated! Please refer to Batsim packages definition in kapack for up-to-date information — in case of doubt, *Contact us*.

```
# Configure the project ; use 'build' as build directory
meson build # --prefix=/desired/installation/prefix

# Actually build batsim
ninja -C build

# Install batsim
meson install -C build
```

CHAPTER 7

Command-line Interface

All usages and options of the command-line interface can be shown by executing `batsim --help`. We present some examples that shows how you can use and combine the different options of Batsim's CLI. All examples suppose that you are locate in the root folder of Batim.

Note that we do not give examples of the scheduler command, as it depends on the *scheduling implementation* you use.

7.1 First run

A simple simulation with a small platform and only one computation job workload as input is done on the Batsim side with the command:

```
batsim -p platforms/small_platform.xml -w workloads/test_one_computation_job.json
```

7.2 Using dynamic jobs

If you want to enable *Dynamic registration of jobs* and profiles by your scheduler during the simulation and you want Batsim to acknowledge registered jobs, you can run:

```
batsim -p platforms/small_platform.xml -w workloads/test_one_computation_job.json \
    --enable-dynamic-jobs --acknowledge-dynamic-jobs
```

7.3 Using Redis

If you want to use *Redis* during your simulation and specify its hostname, you can run:

```
batsim -p platforms/small_platform.xml -w workloads/test_one_computation_job.json \
    --enable-redis --redis-hostname 123.246.0.4
```

7.4 Using external events

If you want to include *External Events* in your simulation with an input file containing external events know by Batsim and an input file containing generic events that you defined you can run:

```
batsim -p platforms/small_platform.xml -w workloads/test_one_computation_job.json \
    --events events/test_events_4hosts.txt \
    --events events/my_generic_events.txt --forward-unknown-events
```

7.5 Example with various options

Finally, if you want to simulate

- a long workload and a workload for energy testing at the same time,
- on the `energy_platform.xml`,
- making host `Mars` a `storage` resource,
- enabling sharing of the compute resources,
- with redis enabled on the specified port `6789`,
- with energy support
- an extra simgrid logging option to set the level of the energy pluggin as critical,
- specifying a folder in the prefix of the output files,
- specifying the socket endpoint to communicate with the scheduler,
- batsim logging on debug level,

you can run:

```
batsim -p platforms/energy_platform.xml -w workloads/test_long_workload.json \
    -w workloads/test_energy_minimal_load100.json \
    --add-role-to-hosts Mars:storage \
    --enable-compute-sharing \
    --enable-redis --redis-port 6789 \
    -E --sg-log surf_energy.thresh:critical \
    --export simu_outputs/out \
    --socket-endpoint ipc://foobar \
    -v debug
```

CHAPTER 8

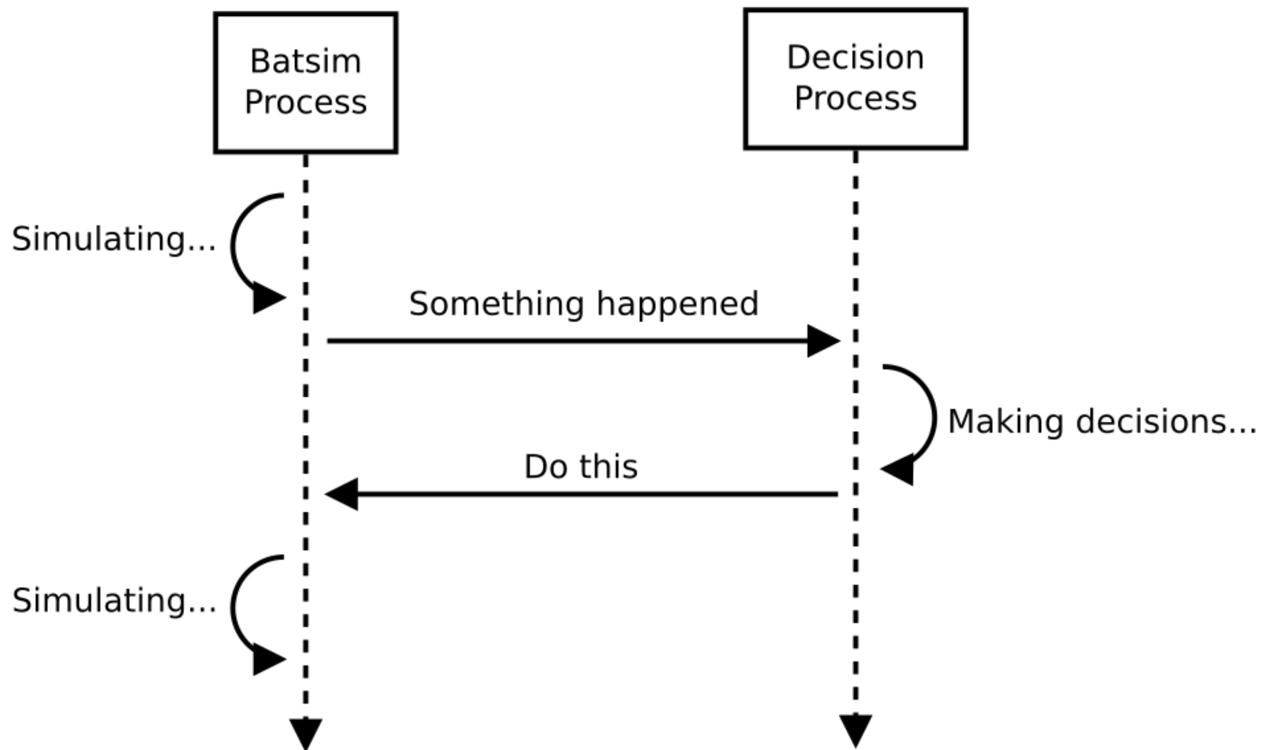
Protocol

A Batsim simulation consists in two processes.

- Batsim itself, in charge of simulating what happens on the platform.
- A *Decision Process* (or more simply scheduler), in charge of making decisions.

The two processes communicate via a socket with the protocol explained in the present document. The protocol is synchronous and follows a simple request-reply pattern. Whenever an event which may require making decision occurs in Batsim in the simulation, the following steps occur.

1. Batsim suspends the simulation
2. Batsim sends a request to the scheduler (telling it what happened on the platform)
3. Batsim waits for a reply from the scheduler
4. Batsim receives the reply
5. Batsim resumes the simulation, applying the decision which have been made



Communication is implemented using the ZeroMQ request-reply pattern. Batsim uses a ZMQ REQ socket to send requests to the schedulers. The scheduler uses a ZMQ REP socket.

The behavior of this protocol depends on Batsim *Command-line Interface*.

- If Redis is enabled, job metadata is stored into a Redis server and not sent through the protocol. In this case, the protocol is only used for synchronization purposes. More information about Redis conventions are described in *Using Redis*.
- Batsim may or may not forward job profile information to the scheduler when jobs are submitted (see *JOB_SUBMITTED* documentation).
- Dynamic jobs (and profile) registration can be enabled or disabled. Many parameters of jobs registration can be adjusted. Please refer to *Dynamic registration of jobs* for more details.

8.1 Message Composition

Each message is a JSON object that looks like this.

```
{
  "now": 1024.24,
  "events": [
    {
      "timestamp": 1000,
      "type": "EXECUTE_JOB",
      "data": {
        "job_id": "workload!job_1234",
        "alloc": "1 2 4-8",
      }
    },
```

(continues on next page)

(continued from previous page)

```
      {
        "timestamp": 1012,
        "type": "EXECUTE_JOB",
        "data": {
          "job_id": "workload!job_1235",
          "alloc": "12-100",
        }
      }
    ]
}
```

The now field defines the current simulation time.

- If the message comes from Batsim, it means that the scheduler cannot make decisions before now as it would change the past.
- If the message comes from the scheduler, it tells Batsim that the scheduler finished making its decisions at timestamp now. This is used by Batsim to know when the scheduler will be available for making new decisions.

The events field defines a sequence of events. The various event types are defined in the present document. See *Table of Events* for a quick list.

8.2 Constraints

Constraints on the message format are defined here.

- The message timestamp now **must** be greater than or equal to every event timestamp.
- Events timestamps **must** be in (non-strictly) ascending order.
- The following fields are mandatory in the message main object.
  - now of float type.
  - events of array type. Can be empty.
    - timestamp of float type.
    - type of string type. Value must be valid (see *Table of Events*).
    - data of object type. Value depends on the event type (see *Table of Events*).

8.3 Table of Events

- Bidirectional
  - *QUERY*
  - *ANSWER*
  - *NOTIFY*
- Batsim to Scheduler
  - *SIMULATION_BEGINS*
  - *SIMULATION_ENDS*
  - *JOB_SUBMITTED*

  - *JOB_COMPLETED*
  - *JOB_KILLED*
  - *RESOURCE_STATE_CHANGED*
  - *REQUESTED_CALL*
- Scheduler to Batsim
  - *REJECT_JOB*
  - *EXECUTE_JOB*
  - *CALL_ME_LATER*
  - *KILL_JOB*
  - *REGISTER_JOB*
  - *REGISTER_PROFILE*
  - *SET_RESOURCE_STATE*
  - *SET_JOB_METADATA*
  - *CHANGE_JOB_STATE*

8.4 Bidirectional events

These events can be sent from Batsim to the scheduler, or in the opposite direction.

8.4.1 QUERY

This event allows a peer to ask specific information to its counterpart. The other peer should answer to such a *QUERY* via an *ANSWER*.

For now, Batsim **answers** to the following requests.

- `consumed_energy`: The scheduler queries Batsim about the total consumed energy (from time 0 to now) in Joules. This query has no argument. Only works if the energy mode is enabled (see *Command-line Interface*).

For now, Batsim **queries** the following requests.

- `estimate_waiting_time`: Batsim asks the scheduler what would be the waiting time of a potential job. Arguments: a job description, similar to those sent in *JOB_SUBMITTED* events when redis is disabled.

data: A dictionary of requests.

```
{
  "timestamp": 10.0,
  "type": "QUERY",
  "data": {
    "requests": {"consumed_energy": {}}
  }
}
```

```
{
  "timestamp": 10.0,
  "type": "QUERY",
```

(continues on next page)

(continued from previous page)


```
  "data": {
    "requests": {
      "estimate_waiting_time": {
        "job_id": "workflow_submitter0!potential_job17",
        "job": {
          "res": 1,
          "walltime": 12.0
        }
      }
    }
  }
}
```


8.4.2 ANSWER

This is a reply to a *QUERY* event.

data: See *QUERY*.


```
{
  "timestamp": 10.0,
  "type": "ANSWER",
  "data": {"consumed_energy": 12500.0}
}
```

```
{
  "timestamp": 10.0,
  "type": "ANSWER",
  "data": {
    "estimate_waiting_time": {
      "job_id": "workflow_submitter0!potential_job17",
      "estimated_waiting_time": 56
    }
  }
}
```


8.4.3 NOTIFY

This event allows a peer to notify something to its counterpart. There is no expected acknowledgment when sending such an event.

For now, Batsim can **notify** the scheduler of the following.

- `no_more_static_job_to_submit`: Batsim tells the scheduler that it has no more jobs to submit from the static submitters. This means that all jobs in the workloads have already been submitted to the scheduler and the scheduler cannot expect more jobs to arrive (except the potential ones through dynamic submission).
- `no_more_external_event_to_occur`: Only applicable if a list of events are given as input to Batsim via the `--events` command-line option. Batsim tells the scheduler that there is no more external event to occur from the event submitters. That means that all external events have occurred and the scheduler cannot expect a new event to occur.
- `event_machine_unavailable` or `event_machine_available` if external events are used (cf. *External Events*).

For now, the scheduler can **notify** Batsim of the following.

- `registration_finished`: The scheduler tells Batsim that dynamic job registrations are over, therefore allowing Batsim to stop the simulation eventually. This event **MUST** be sent if dynamic jobs registration is enabled (see *Command-line Interface*).
- `continue_registration`: The scheduler tells Batsim that it has sent a `registration_finished` *NOTIFY* prematurely and that Batsim should re-enable dynamic registration of jobs...

data: The type of notification, as a string.

```
{
  "timestamp": 23.50,
  "type": "NOTIFY",
  "data": { "type": "no_more_static_job_to_submit" }
}
```

```
{
  "timestamp": 23.50,
  "type": "NOTIFY",
  "data": { "type": "no_more_external_event_to_occur" }
}
```

```
{
  "timestamp": 42.0,
  "type": "NOTIFY",
  "data": { "type": "registration_finished" }
}
```

```
{
  "timestamp": 42.0,
  "type": "NOTIFY",
  "data": { "type": "continue_registration" }
}
```

8.5 Batsim to Scheduler events

These events are sent by Batsim to the scheduler.

8.5.1 SIMULATION_BEGINS

Sent at the beginning of the simulation. If Redis is enabled, the scheduler can read metainformation from the Redis server as soon as *SIMULATION_BEGINS* has been received.

Batsim configuration is sent through the `config` object (in `data`). This object contains some of Batsim's options as chosen by user at runtime (see *Command-line Interface*). In particular, if a scheduler configuration is set via Batsim's *Command-line Interface*, it is forwarded to the scheduler in the `sched-config` string inside the `config` object.

data: An object with the following fields.

- `nb_resources`: The number of resources in the simulated platform.
- `nb_compute_resources`: The number of compute resources in the simulated platform.
- `nb_storage_resources`: The number of storage resources in the simulated platform.

- `allow_compute_sharing`: Whether compute hosts can be used at the same time by several jobs or not (see *Command-line Interface*).
- `allow_storage_sharing`: Whether storage hosts can be used at the same time by several jobs or not (see *Command-line Interface*).
- `config`: The Batsim configuration.
- `compute_resources`: Information about the compute resources.
- `id`: Unique resource number.
- `name`: Resource name.
- `state`: Resource state in {`sleeping`, `idle`, `computing`, `switching_on`, `switching_off`}.
- `properties`: The properties specified in the SimGrid platform for the corresponding host.
- `storage_resources`: Information about the storage resources.
- `workloads`: The object of the workloads given to Batsim. The key is the unique id of the workload and the value is the absolute path of the workload. Note that this unique id prefixes each job (before the `!`).
- `profiles`: The object of profiles given to Batsim. The key is the unique id of the workload and the value is the list of profiles of that workload.

```
{
  "now": 0,
  "events": [
    {
      "timestamp": 0,
      "type": "SIMULATION_BEGINS",
      "data": {
        "nb_resources": 4,
        "nb_compute_resources": 4,
        "nb_storage_resources": 0,
        "allow_compute_sharing": false,
        "allow_storage_sharing": true,
        "config": {
          "redis-enabled": false,
          "redis-hostname": "127.0.0.1",
          "redis-port": 6379,
          "redis-prefix": "default",
          "profiles-forwarded-on-submission": false,
          "dynamic-jobs-enabled": false,
          "dynamic-jobs-acknowledged": false,
          "profile-reuse-enabled": false,
          "sched-config": "Scheduler-specific configuration. In this instance, just a␣
↪meaningless example string.",
          "forward-unknown-events": false
        },
        "compute_resources": [
          {
            "id": 0,
            "name": "Bourassa",
            "state": "idle",
            "properties": {
              "role": ""
            },
            "zone_properties": {}
          },
```

(continues on next page)

(continued from previous page)

```
        {
          "id": 1,
          "name": "Fafard",
          "state": "idle",
          "properties": {
            "role": ""
          },
          "zone_properties": {}
        },
        {
          "id": 2,
          "name": "Ginette",
          "state": "idle",
          "properties": {
            "role": ""
          },
          "zone_properties": {}
        },
        {
          "id": 3,
          "name": "Jupiter",
          "state": "idle",
          "properties": {
            "role": ""
          },
          "zone_properties": {}
        }
      ],
      "storage_resources": [],
      "workloads": {
        "w0": "/home/carni/proj/batsim/workloads/test_various_profile_types.json"
      },
      "profiles": {
        "w0": {
          "homogeneous_total": {
            "type": "parallel_homogeneous_total",
            "cpu": 10000000,
            "com": 1000000
          },
          "homogeneous": {
            "type": "parallel_homogeneous",
            "cpu": 10000000,
            "com": 1000000
          },
          "simple": {
            "type": "parallel",
            "cpu": [
              5000000,
              0,
              0,
              0
            ],
            "com": [
              5000000,
              0,
              0,
              0,
```

(continues on next page)

(continued from previous page)

```
                5000000,
                5000000,
                0,
                0,
                5000000,
                5000000,
                0,
                0,
                5000000,
                5000000,
                5000000,
                0
              ]
            },
            "homogeneous_no_com": {
              "type": "parallel_homogeneous",
              "cpu": 200000,
              "com": 0
            },
            "homogeneous_no_cpu": {
              "type": "parallel_homogeneous",
              "cpu": 0,
              "com": 1000000
            },
            "sequence": {
              "type": "composed",
              "repeat": 4,
              "seq": [
                "simple",
                "homogeneous",
                "simple"
              ]
            },
            "delay": {
              "type": "delay",
              "delay": 20.2
            }
          }
        }
      }
    }
  ]
}
```

8.5.2 SIMULATION_ENDS

Sent when Batsim thinks that the simulation is over. It means that all the jobs (either coming from Batsim workloads/workflows inputs, or dynamically submitted) have been submitted and executed (or rejected).

When receiving a *SIMULATION_ENDS*, the scheduler should answer a message without events, close its socket then terminate.

data: None.

```
{
  "timestamp": 100.0,
  "type": "SIMULATION_ENDS",
  "data": {}
}
```

8.5.3 JOB_SUBMITTED

The content of this event depends on how Batsim has been called (see *Command-line Interface*).

This event means that one job has been submitted within Batsim. It is sent whenever a job coming from Batsim inputs (workloads and workflows) has been submitted. If dynamic jobs registration is enabled, this event is sent as a reply to a *REGISTER_JOB* event if and only if dynamic jobs registration acknowledgments are also enabled. More information can be found in *Dynamic registration of jobs*.

The `job_id` field is always sent and contains a unique job identifier. If redis is enabled, `job_id` is the only forwarded field. Otherwise (i.e., if redis is disabled), a JSON description of the job is forwarded in the `job` field.

A JSON description of the job profile is sent if and only if profiles forwarding is enabled (see *Command-line Interface*).

data: a job id and optional information depending on how Batsim has been called (see *Command-line Interface*).

Example **without redis and without forwarded profiles**.

```
{
  "timestamp": 10.0,
  "type": "JOB_SUBMITTED",
  "data": {
    "job_id": "dyn!my_new_job",
    "job": {
      "profile": "delay_10s",
      "res": 1,
      "id": "dyn!my_new_job",
      "walltime": 12.0
    }
  }
}
```

Example **without redis and with forwarded profiles**.

```
{
  "timestamp": 10.0,
  "type": "JOB_SUBMITTED",
  "data": {
    "job_id": "dyn!my_new_job",
    "job": {
      "profile": "delay_10s",
      "res": 1,
      "id": "dyn!my_new_job",
      "walltime": 12.0
    },
    "profile":{
      "type": "delay",
      "delay": 10
    }
  }
}
```

Example **with redis**.


```
{
  "timestamp": 10.0,
  "type": "JOB_SUBMITTED",
  "data": {"job_id": "w0!1"}
}
```


8.5.4 JOB_COMPLETED

This event means that a job has completed its execution. It acknowledges that the actions coming from a previous *EXECUTE_JOB* event have been done (successfully or not, depending on whether the job completed without reaching timeout).

data: An object with the following fields.

- `job_id`: The job unique identifier.
- `job_state`: The job state. Possible values: `NOT_SUBMITTED`, `SUBMITTED`, `RUNNING`, `COMPLETED_SUCCESSFULLY`, `COMPLETED_FAILED`, `COMPLETED_WALLTIME_REACHED`, `COMPLETED_KILLED`, `REJECTED`.
- `return_code`: The return code of the job process (equals to 0 by default, see *Workload*).
- `alloc`: The *Interval set* of resources allocated to this job in the previous *EXECUTE_JOB* event.


```
{
  "timestamp": 80.087881,
  "type": "JOB_COMPLETED",
  "data": {
    "job_id": "26dceb!4",
    "job_state": "COMPLETED_SUCCESSFULLY",
    "return_code": 0,
    "alloc": "0-3"
  }
}
```


8.5.5 JOB_KILLED

This event means that some jobs have been killed. It acknowledges that the actions coming from a previous *KILL_JOB* event have been done. The `job_ids` correspond to those requested in the previous *KILL_JOB* event.

The `job_progress` object is also given for all the jobs (and for the tasks inside the jobs) that have been killed. Key is the `job_id` and the value contains a progress value in]0, 1[, where 0 means not started and 1 means completed. The profile name is also given for convenience. For jobs with a SEQUENCE profile, the progress map contains the 0-based index of the inner task that was running at the time it was killed, and the details of this progress are in the `current_task` field. Please note that sequential jobs can be nested.

Please remark that this event does not necessarily mean that all the jobs have been killed. It means that all the jobs have completed. Some of the jobs might have completed *ordinarily* before the kill. In this case, *JOB_COMPLETED* events corresponding to the aforementioned jobs should be received before the *JOB_KILLED* event.

data: A list of job ids + progress of the jobs that have been killed.

Example **without progress**. In this case, none of the jobs have really been killed — they finished *ordinarily* before the kill.

```
{
  "timestamp": 10.0,
  "type": "JOB_KILLED",
  "data": {
    "job_ids": [
      "w0!1",
      "w0!2"
    ]
  }
}
```


Example **with progress**. In this case, the three jobs have really been killed. Job `w0!1` has been killed after computing 52 % of the whole job (`progress` is 0.52). As job `w0!2` uses a composed profile (a sequence), its progress is a bit more complex. `w0!2` has been killed during the second task of the sequence (`current_task_index` is 1 and task indexes are 0-based), and 20 % of that task has been done (`progress` is 0.2). Job `w0!3` uses a nested sequence of profiles. It has been stopped during the third (`current_task_index` is 2) task of the root sequence, which is itself a sequence. This sequence has been stopped during the fourth task (`current_task_index` is 3), which is a simple profile whose 75 % of the work (`progress` is 0.75) could be done before the kill.


```
{
  "timestamp": 10.0,
  "type": "JOB_KILLED",
  "data": {
    "job_ids": [
      "w0!1",
      "w0!2",
      "w0!3"
    ],
    "job_progress": {
      "w0!1": {
        "profile": "my_simple_profile",
        "progress": 0.52
      },
      "w0!2": {
        "profile": "my_sequence_profile",
        "current_task_index": 1,
        "current_task": {
          "profile": "my_simple_profile",
          "progress": 0.2
        }
      },
      "w0!3": {
        "profile": "my_sequence_of_sequences_profile",
        "current_task_index": 2,
        "current_task": {
          "profile": "my_sequential_profile",
          "current_task_index": 3,
          "current_task": {
            "profile": "my_simple_profile",
            "progress": 0.75
          }
        }
      }
    }
  }
}
```

8.5.6 RESOURCE_STATE_CHANGED

This event means that the state of some resources has changed. It acknowledges that the actions coming from a previous *SET_RESOURCE_STATE* event have been done.

data: An *Interval set* of `resources` and their new `state`.

```
{
  "timestamp": 10.0,
  "type": "RESOURCE_STATE_CHANGED",
  "data": {"resources": "1 2 3-5", "state": "42"}
}
```

8.5.7 REQUESTED_CALL

This event is a response to the *CALL_ME_LATER* event.

data: None.

```
{
  "timestamp": 25.5,
  "type": "REQUESTED_CALL",
  "data": {}
}
```

8.6 Scheduler to Batsim events

These events are sent by the scheduler to Batsim.

8.6.1 REJECT_JOB

Rejects a job that has already been submitted. The rejected job will not appear into the final jobs trace.

data: A job id.

```
{
  "timestamp": 10.0,
  "type": "REJECT_JOB",
  "data": { "job_id": "w12!45" }
}
```

8.6.2 EXECUTE_JOB

Execute a job on a given *Interval set* of resources.

An optional `mapping` field can be added to tell Batsim how to map executors to resources: Where the executors will be placed inside the allocation (resource numbers are shifted to 0). It can be seen as MPI rank to host mapping. The following example overrides the default round robin mapping to put the first two ranks (0 and 1) on the first allocated machine (0, which stands for resource id 2), and the last two ranks (2 and 3) on the second machine (1, which stands for resource id 3).

For certain job profiles that involve storage you may need to define a `storage_mapping` between the storage label defined in the job profile definition and the storage resource id of the platform. For example, the job profile of type `parallel_homogeneous_pfs` contains this field `"storage": "pfs"`. In order to select what is the resource that corresponds to the `"pfs"` storage, you should provide a mapping for this label: `"storage_mapping": { "pfs": 2 }`. If no mapping is provided, Batsim will guess the storage mapping only if one storage resource is provided on the platform. For the case of `data_staging` profile, the `storage_mapping` is mandatory and must specify the resource id of the storages associated to the `to` and `from` labels of the profile.

Another optional field is `additional_io_job` that permits the scheduler to add a job, that represents the IO traffic, dynamically at execution time. This dynamicity is necessary when the IO traffic depends on the job allocation. It only works for parallel task based job profile types for the additional IO job and the job itself. The given IO job will be merged to the actual job before its execution. The additional job allocation may be different from the job allocation itself, for example when some IO nodes are involved.

data: A job id, an allocation of resources `alloc` (see *Interval set string representation* for format), a mapping (optional), an additional IO job (optional).

```
{
  "timestamp": 10.0,
  "type": "EXECUTE_JOB",
  "data": {
    "job_id": "w12!45",
    "alloc": "2-3",
    "mapping": {"0": "0", "1": "0", "2": "1", "3": "1"},
    "storage_mapping": {
      "pfs": 2
    },
  },
  "additional_io_job": {
    "alloc": "2-3 5-6",
    "profile_name": "my_io_job",
    "profile": {
      "type": "parallel",
      "cpu": 0,
      "com": [0  ,5e6,5e6,5e6,
              5e6,0  ,5e6,0  ,
              0  ,5e6,4e6,0  ,
              0  ,0  ,0  ,0  ]
    }
  }
}
```

8.6.3 CALL_ME_LATER

Asks Batsim to call the scheduler later on, at a given timestamp.

Batsim will send a *REQUESTED_CALL* event when the desired timestamp is reached.

data: When the scheduler desires to be called.

```
{
  "timestamp": 10.0,
  "type": "CALL_ME_LATER",
  "data": {"timestamp": 25.5}
}
```

8.6.4 KILL_JOB

Kill some jobs (almost instantaneously).

As soon as all the jobs defined in the `job_ids` field have completed (most probably killed, but they may also have finished *ordinarily* before the kill), Batsim acknowledges it with one *JOB_KILLED* event.

data: A list of job ids.

```
{
  "timestamp": 10.0,
  "type": "KILL_JOB",
  "data": {"job_ids": ["w0!1", "w0!2"]}
}
```

8.6.5 REGISTER_JOB

Registers a job (from the scheduler) at the current simulation time.

Jobs registration from the scheduler must be enabled (see *Command-line Interface*). Acknowledgment of registrations can be enabled (see *Command-line Interface*). More information can be found in *Dynamic registration of jobs*.

Important note: The workload name SHOULD be present in the job description id field with the notation `WORKLOAD!JOB_NAME`. If it is not present it will be added to the job description provided in the acknowledgment message *JOB_SUBMITTED*.

data: A job id (job id duplication is forbidden), classical job and profile information (optional).

Example **without redis** : The whole job description goes through the protocol.

```
{
  "timestamp": 10.0,
  "type": "REGISTER_JOB",
  "data": {
    "job_id": "dyn!my_new_job",
    "job":{
      "profile": "delay_10s",
      "res": 1,
      "id": "dyn!my_new_job",
      "walltime": 12.0
    }
  }
}
```

Example **with redis** : The job and profile description, if unknown to Batsim yet, must have been pushed into the Redis server by the scheduler before sending this message. See *Using Redis*.

```
{
  "timestamp": 10.0,
  "type": "REGISTER_JOB",
  "data": {
    "job_id": "w12!45",
  }
}
```

8.6.6 REGISTER_PROFILE

Registers a profile (from the scheduler).

Jobs registration from the scheduler must be enabled (see *Command-line Interface*). More information can be found in *Dynamic registration of jobs*.

data: A workload name, profile name, and the data of the profile.

Example **without redis** : The whole profile description goes through the protocol.

```
{
  "timestamp": 10.0,
  "type": "REGISTER_PROFILE",
  "data": {
    "workload_name": "dyn_wl1",
    "profile_name":  "delay_10s",
    "profile": {
      "type": "delay",
      "delay": 10
    }
  }
}
```

With redis : Instead of using this event, the profiles should be pushed to redis directly by the scheduler.

8.6.7 SET_RESOURCE_STATE

Sets some resources into a state.

As soon as all the resources have been set into the given state, Batsim acknowledges it by sending one *RESOURCE_STATE_CHANGED* event.

data: An *Interval set* of `resources` and their new `state`.

```
{
  "timestamp": 10.0,
  "type": "SET_RESOURCE_STATE",
  "data": {"resources": "1 2 3-5", "state": "42"}
}
```

8.6.8 SET_JOB_METADATA

A convenient way to attach metadata to a job during simulation runtime that will appear in the final result file. A column named `metadata` will be present in the output file `PREFIX_job.csv` with the string provided by the scheduler, or an empty string if not set.

Note: If you need to add **static** metadata to a job you can simply add one or more fields in the job profile.

Warning: This not a way to delegate to Batsim the storage of metadata. That should be done through Redis (when you have to share information between different processes for example), or using the scheduler's internal data structures.

data: A job id and its metadata.

```
{
  "timestamp": 13.0,
  "type": "SET_JOB_METADATA",
```

(continues on next page)

(continued from previous page)

```
  "data": {
    "job_id": "wload!42",
    "metadata": "scheduler-defined string"
  }
}
```

8.6.9 CHANGE_JOB_STATE

Changes the state of a job, which may be helpful to implement schedulers with complex dynamic jobs.

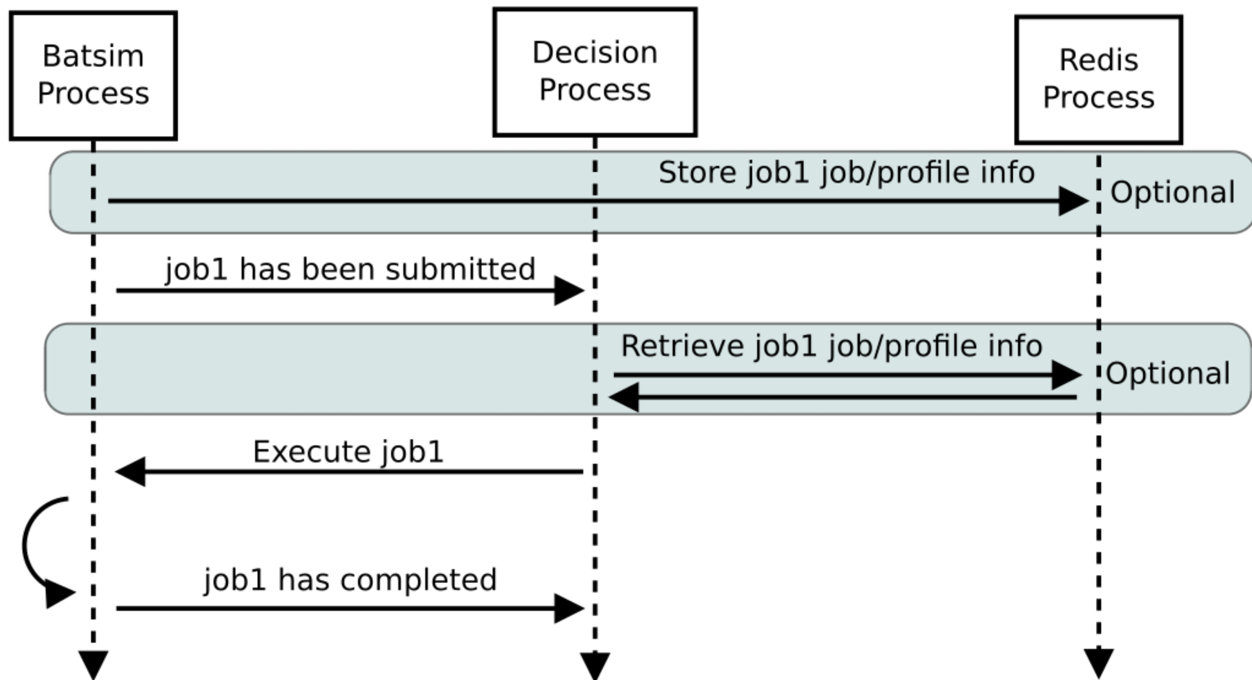
```
{
  "timestamp": 42.0,
  "type": "CHANGE_JOB_STATE",
  "data": {
    "job_id": "w12!45",
    "job_state": "COMPLETED_KILLED",
    "kill_reason": "Sub-jobs were killed."
  }
}
```

8.7 Figuration of common scenarios

The way to do some operations with the protocol is shown in this section.

8.8 Executing jobs

Depending on how Batsim is called (see *Command-line Interface*), jobs information might either be transmitted through the protocol or Redis.



8.9 Dynamic registration of jobs

Jobs are in most cases given as Batsim inputs, which are submitted within Batsim (the scheduler knows about them via *JOB_SUBMITTED* events).

However, jobs can also be submitted from the scheduler (via registration events) throughout the simulation. For this purpose:

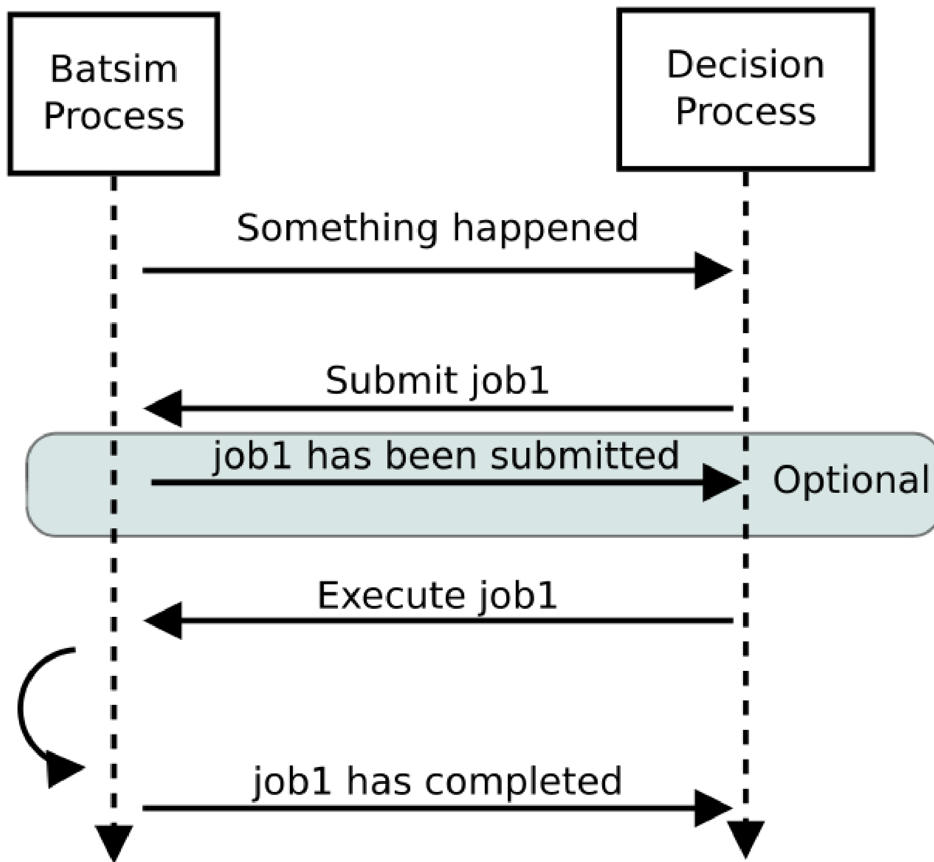
- Dynamic jobs registration **must** be enabled (see *Command-line Interface*).
- The scheduler **must** tell Batsim when it has finished registering dynamic jobs (via a *NOTIFY* event). Otherwise, Batsim will wait for new simulation events forever, causing either a SimGrid deadlock or an infinite loop at the end of the simulation.
- the scheduler **must** make sure that Batsim has enough information to avoid SimGrid deadlocks during the simulation. If at some simulation time all Batsim workloads/workflows inputs have been executed and nothing is happening on the platform, this might lead to a SimGrid deadlock. If the scheduler knows that it will register a dynamic job in the future, it should ask Batsim to call it at this timestamp via a *CALL_ME_LATER* event.

The protocol behavior of dynamic registrations is customizable (see *Command-line Interface*). - Batsim might or might not send acknowledgments when jobs have been registered. - Metainformation are sent via Redis if Redis is enabled, or directly via the protocol otherwise.

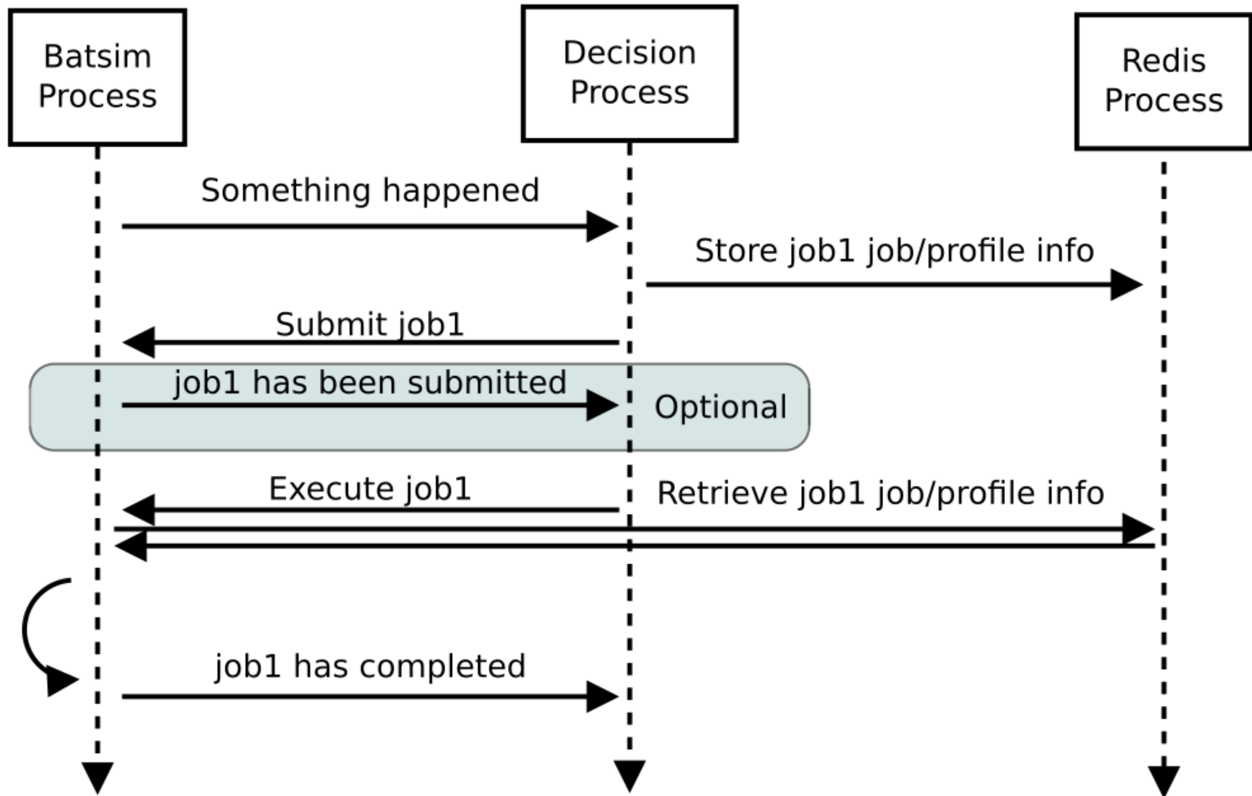
A simple scheduling algorithm using dynamic jobs registration can be found in the batsched submitter algorithm. This implementation should work whether Redis is enabled and whether dynamic job registrations are acknowledged.

The following two figures outline how registrations should be done (depending on whether Redis is enabled or not).

8.9.1 Without Redis



8.9.2 With Redis

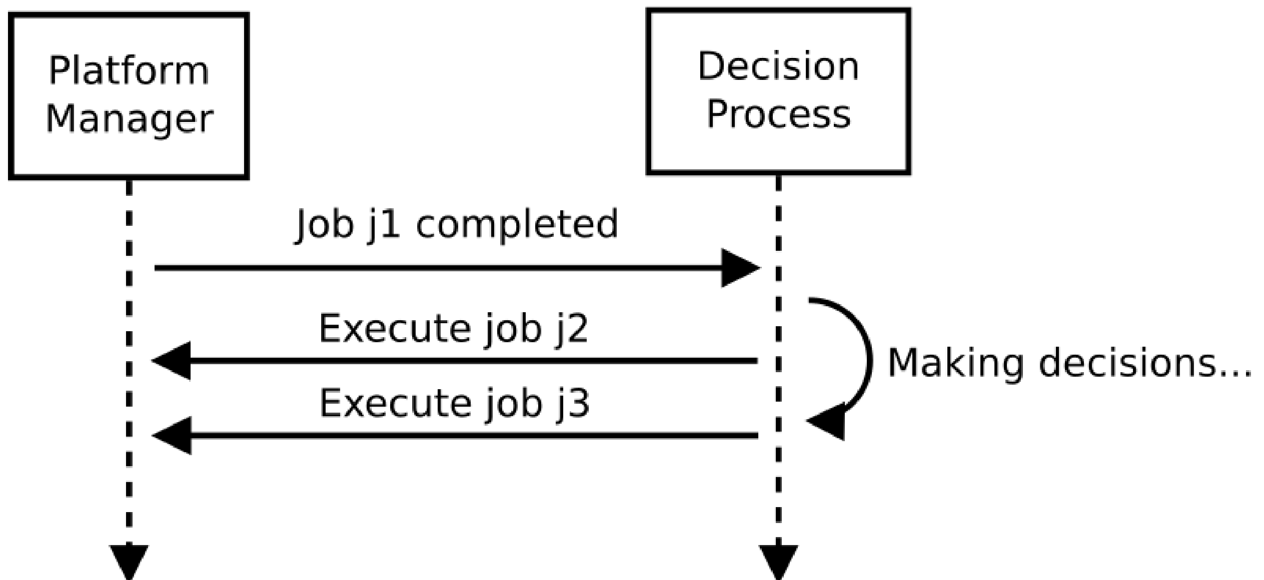


CHAPTER 9

Progression of simulation time

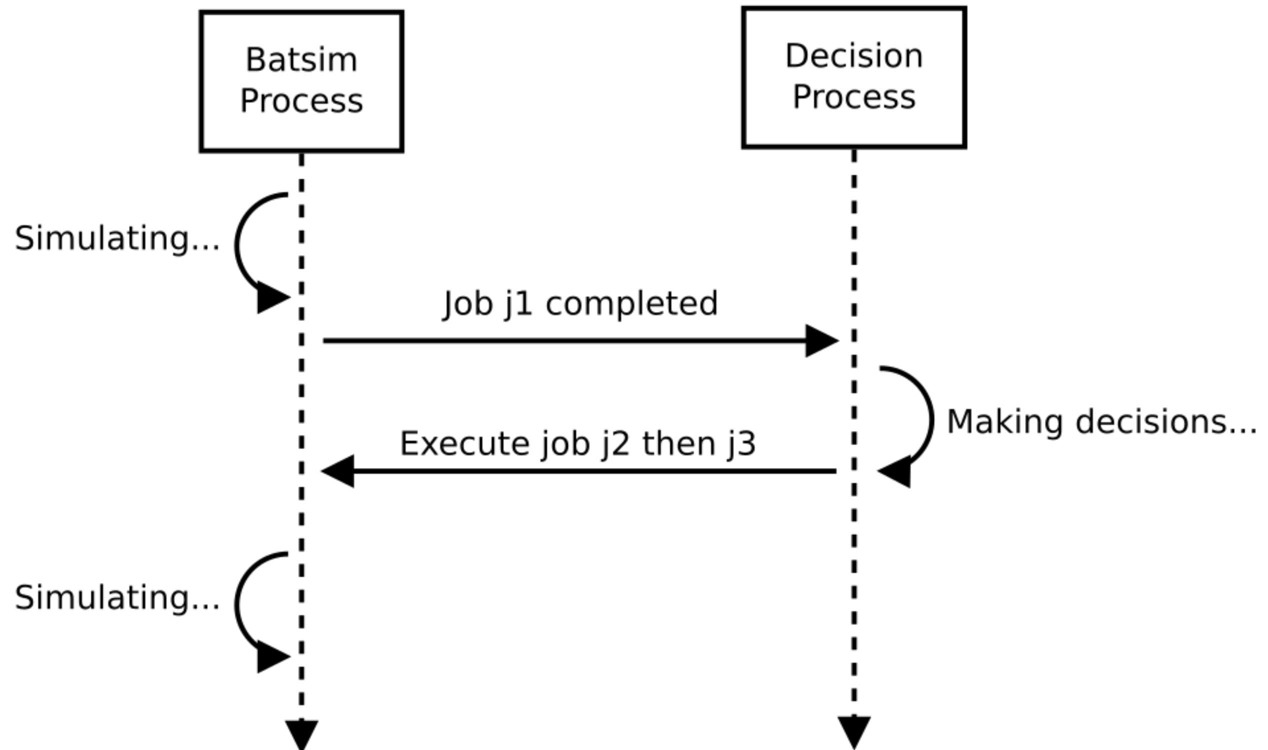
In a real system, resource management procedures are called from time to time to take some decisions. This example shows how the simulation time is managed in Batsim. It stresses how decision making time can be taken into account in the simulation.

This case study consists in taking the decision to execute two jobs (`j2` and `j3`) when job `j1` completes. The important point here is that the decision-making procedure takes some time and takes the decision in an online fashion: The decision to execute `j2` is made before the decision to execute `j3`, and here we want to inject the scheduling time into the simulation.



9.1 Message exchange overview

As a reminder, a Batsim simulation involves two processes that communicate via the Batsim *Protocol*. Here is a figuration of this case study.



Concretely, the message sent from Batsim to the Scheduler looks like this.

```
{
  "now": 10.000000,
  "events": [
    {
      "timestamp": 10.000000,
      "type": "JOB_COMPLETED",
      "data": {
        "job_id": "w0!1",
        "status": "SUCCESS"
      }
    }
  ]
}
```

- At time `10`, job `1` (from workload `w0`) completed successfully.
- The scheduler has been called at time 10 (`"now": 10.000000`). It means that the decisions can be taken at time 10 **or later on**.

The message replied by the Scheduler looks like this.

```
{
  "now": 15.0,
  "events": [
    {
```

(continues on next page)

(continued from previous page)

```
      "timestamp": 13.0,
      "type": "EXECUTE_JOB",
      "data": {
        "job_id": "d4c32e!2",
        "alloc": "0-1"
      }
    },
    {
      "timestamp": 14.0,
      "type": "EXECUTE_JOB",
      "data": {
        "job_id": "d4c32e!3",
        "alloc": "2-3"
      }
    }
  ]
}
```

This message means that the scheduler:

- (Did something until time `13`, since the request has been sent at time `10` and that the first event is at time `13`.)
- First chose, at time `13`, to execute job `2` on machines `0` and `1`.
- (Did something until time `14`, since the next event is at time `14`.)
- Then chose, at time `14`, to execute job `3` on machines `2` and `3`
- (Did something until time 15, since the reply has been received at `"now": 15.0`)
- Finally chose to stop taking decisions for now, at time `15`.

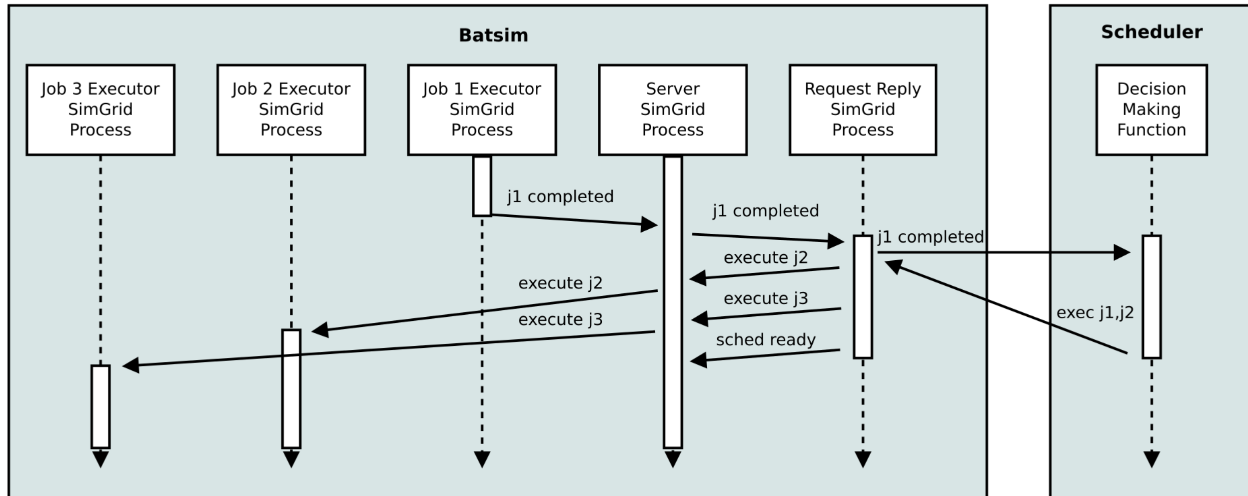
9.2 What happens within Batsim?

Batsim can be seen as a distributed application composed of different processes. These processes may communicate with each other, and spawn other processes.

The main process is the **server**. It is started at the beginning of the simulation, and it ends when the simulation has finished. It orchestrates most of the other processes:

- **request reply** processes, in charge of communicating with the scheduler.
- **job executor** processes, in charge of executing jobs.
- **waiter** processes, in charge of handling *CALL_ME_LATER* events.
- etc.

What happens within Batsim for the case study 1 is the following.



First, a **job executor** process finishes to execute job `j1`. It sends a message about it to the **server** then terminates. When the server receives the message, it spawns a **request reply** process to forward that `j1` has completed.

The newly spawned **request reply** process sends a network message to the scheduler, forwarding that `j1` has completed. The **request reply** process then waits for the scheduler's reply: **The simulation is *stopped* as long as the reply has not been received**. Once the reply from the scheduler has been received, the **request reply** process role is to forward the events to the server at the right times. For this purpose, it sends the events in order, sleeping between events if needed.

Once all the events have been forwarded, the **request reply** process sends a `SCHED_READY` message to the **server**. This message means that all the events coming from the scheduler have been sent, and that the scheduler is now ready to be called if needed.

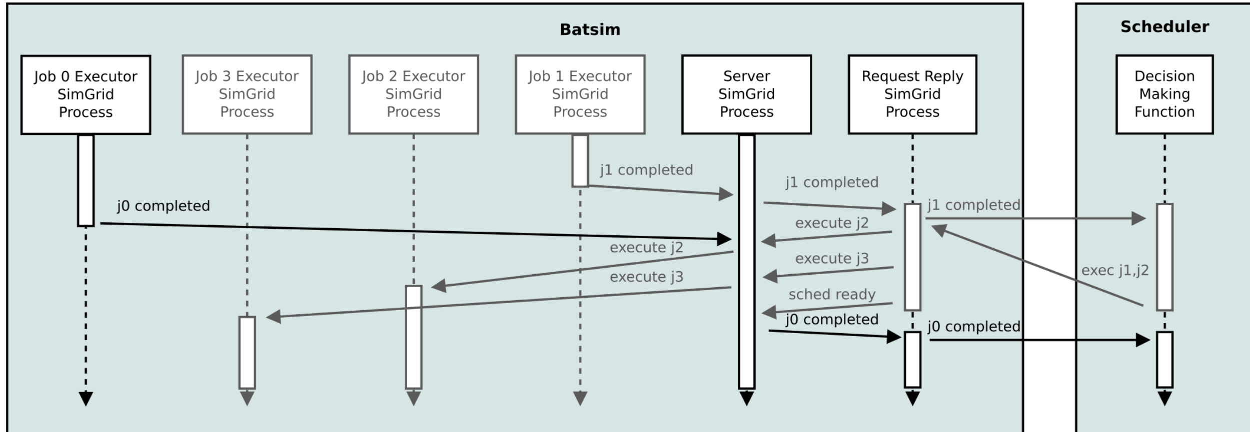
Events received by the **server** that must be forwarded to the scheduler are queued in a data structure kept in memory. If the scheduler is ready, the queued event is sent immediately. Otherwise, the queued events will be sent as soon as possible — i.e., when the next `SCHED_READY` event will have been received. This mechanism ensures that scheduler calls are consistent in time.

- If the scheduler replied at time `t` (`now` field of the reply message), the next call to the scheduler is ensured to occur at a time greater than or equal to `t`.
- When the scheduler is called, it is sure that all its previous decisions have been initiated.

9.3 What if something happened during the scheduler call?

Please remark that this mechanism implies that schedulers — that wish to take scheduling time into account — may receive messages from the *past* when they are called. Batsim can indeed send messages whose events occurred between the last call time (`now` field of the previous request message sent by Batsim) and the current one (`now` field of the current request message).

For example, imagine the same scenario as before but with a job `j0` that finishes at time `13.1`. The scheduler is taking decisions at this time (until time `15`). Hence, the scheduler will finish its decision-making procedure and then be called as follows.



The Batsim request message would look like this.

```
{
  "now": 15.001000,
  "events": [
    {
      "timestamp": 13.100000,
      "type": "JOB_COMPLETED",
      "data": {
        "job_id": "d4c32e!0",
        "status": "SUCCESS"
      }
    }
  ]
}
```

- Job `j0` finished at time `13.1` successfully.
- The current time is `15.001`. Therefore, the scheduler can only take decisions at time `15.001` or afterwards.

CHAPTER 10

Using Redis

By default, Batsim's *Protocol* transfers job metadata directly on the socket between Batsim and the *Decision Process*. Alternatively, you can use a Redis server to transfer this information. Redis support is enabled by providing the `--enable-redis` *Command-line Interface* flag to the Batsim command.

> **Warning:** Redis should also be configured in your *Scheduler implementation*. Redis information is forwarded to the scheduler in the *SIMULATION_BEGINS protocol event*.

10.1 Key prefix

Since several Batsim instances can be run at the same time, all the keys explained in this document must be prefixed by some instance-specific prefix. The key prefix can be set by the `--redis-prefix` *Command-line Interface* option. Batsim will write and read keys by prefixing them by the user-given prefix followed by a colon `:`.

10.2 List of used keys

10.2.1 Job information

As soon as job `<ID>` is submitted within Batsim, the `job_<ID>` key is set. Its value is the JSON description of the job (cf. *Job definition*).

> **Note:** When using Redis with *Dynamic registration of jobs*, Batsim expects to find job information in the Redis store when it receives a *REGISTER_JOB* event.

10.2.2 Profile information

If profile forwarding is enabled (`--forward-profiles-on-submission` *Command-line Interface* option, forwarded in *SIMULATION_BEGINS*), the profile of jobs is also set in the Redis store at job submission time. If the submitted job uses the `<PROF>` profile, the `profile_<PROF>` key is set. Its value is the JSON description of the profile (cf. *Profile definition*).

Note: When using Redis with *Dynamic registration of jobs*, Batsim expects to find profile information in the Redis store when it receives a *REGISTER_PROFILE* event.

CHAPTER 11

File System model

11.1 Storage

The model chosen for the storage is to use a simple SimGrid host, linked to a compute node for local disk, or anywhere in the topology for centralized storage. The storage access latency and read/write bandwidth is modeled by two SimGrid links that attached the storage node to the rest of the platform. This model allows to simulate any kind of file system topology, from classical DFS, by adding a storage to every node of the platform, to centralized storage with a single large storage node, and including any mix of the two.

Here is an example of storage node added to the Taurus platform that represents an HDD:

```
<host id="taurus-16.lyon.grid5000.fr_disk" speed="0">
  <prop id="role" value="storage"/>
</host>
<link bandwidth="150MBps" id="taurus-16.lyon.grid5000.fr_IO_read" latency="0.11ms"/>
<link bandwidth="80MBps" id="taurus-16.lyon.grid5000.fr_IO_write" latency="0.11ms"/>
```

11.2 I/O transfer

Now that the platform contains storage node, we need to model the I/O transfer between them. Batsim implementation of I/O transfer are based on the parallel task model described above. Every storage get a resource ID that identify them in the platform with a unique number, just like the computation resources. Using the storage ID and the normal resource ID, data movement is simply modeled by a matrix of communication involving computation node and storage nodes.

The genericity of the model is ensured by keeping it outside of Batsim. Thus, the burden of maintaining the file system model is delegated to the user's defined process that is connected to Batsim called the decision process, also called the scheduler.

The scheduler uses dynamic job to model file system internal moves like DFS load balance, or data staging from PFS to Burst Buffers, but most of I/O transfer are triggered by applications. But the application model also need to be independent from the file system. So the application model that is defined by the Batsim job profiles, provided to

Batsim with the workload file, needs to contain enough information for the scheduler to generate actual I/O transfer at execution time. Thus, the application model can be statically defined in the Batsim workload file, but the I/O that is triggered by this application is dynamically generated depending on the context by the scheduler.

Concretely, when a job is submitted in the workload, Batsim informs the scheduler of this event with a job profile attached. This job profile contains information about the quantity of I/O that need to be done by the job, or any arbitrary information required by the file system model. Then, the scheduler can choose where the application will be allocated, and also, what are the I/O transfers that will occur during the execution of this application. Once the decision is taken, the scheduler send the job allocation to Batsim inside the job execution order as usual, but includes also an additional I/O job. This I/O job has its own allocation and the associated communication matrix that represent the transfer between the compute nodes of the jobs and the storages. Finally, Batsim will merge this I/O job profile with the normal job profile before to delegate the actual simulation to SimGrid.

11.3 Full example

Todo: Add more detailed to the example:

- the platform file
- the full workload file
- the launch command

Here is an example that illustrate this mechanism. Let's assume that the `job1` was submitted to the scheduler with a request of 2 computing resources. The profile of the job called is defined by a Json object, `job1_profile`.

1. The scheduler receive the submission job from Batsim with this forwarded profile, where the quantity of I/O is just an information given to the scheduler (not read by Batsim):

```
"job1_profile": {
    "type": "parallel_homogeneous_total",
    "cpu": 1e10,
    "com": 1e7,
    "io_reads":  8e8,
    "io_writes": 2e8
}
```

2. The scheduler takes the decision to allocate this job on the resources 0 and 1. Then it decides, regarding its file system model, that each node will read half of the data from only one centralized storage (located at the resource id 10), but the nodes of the application will write on local their local disks (id 2 and 3). A new profile of type `parallel` is generated an submitted to Batsim with the name `io_for_this_alloc_on_job1`. The kind job profile is composed of a computation matrix that may reflect I/O related computation (here set to 0), and a communication matrix that represents the aforementioned assumptions. Note that the direction of the transfer is from row to column, and that the indices of the matrix are mapped to the IO allocation, e.g. for the allocation of $(0, 1, 2, 3, 10)$ a value of $4e8$ on the 1st column and at the 5th row means that 400MB will be transferred from the resource 10 to the resource 0.

```
"io_for_this_alloc_on_job1": {
    "type": "parallel",
    "cpu": [0   ,0   ,0   ,0   ,0]
    "com": [0   ,0   ,1e8,0   ,0
    0   ,0   ,0   ,1e8,0
    0   ,0   ,0   ,0   ,0
    0   ,0   ,0   ,0   ,0
```

(continues on next page)

(continued from previous page)

```
    4e8,4e8,0   ,0   ,0]
}
```

3. The scheduler ask Batsim to execute the job with the given job allocation, and the additional I/O job.

```
{
    "job_id":"08a582!1",
    "alloc":"0-1",
    "additional_io_job": {
        "alloc":"0-3,10",
        "profile": "io_for_this_alloc_on_job1"
    }
}
```

4. Batsim merges the 2 profiles, and generates a new job with IO matrix that is then sent to SimGrid in order to be simulated:

```
{
    "alloc": "0-3,10",
    "cpu": [5e9,5e9,0   ,0   ,0],
    "com": [0   ,5e6,1e8,0   ,0
            5e6,0   ,0   ,1e8,0
            0   ,0   ,0   ,0   ,0
            0   ,0   ,0   ,0   ,0
            4e8,4e8,0   ,0   ,0]
}
```

Note the difference of allocation between the job itself and the IO that it generates. Batsim is capable to merge any interval set of resource allocation, even if only part of the job's nodes are taking part in the IO transfer.

This simple file system model is generic enough to simulate any centralized and decentralized file system, because it not assume any kind of I/O behavior. For example, it is possible to simulated hierarchical file system like a PFS with I/O nodes, or a multi-tiers storage setup with two different centralized file system, e.g. NFS and Lustre, or even a mix of DFS and PFS.

It is fully dynamic: the I/O transfer inside the application are generated online by the decision process, which allows to take the I/O into account for any decision, from job allocation, to I/O gateway selection. Also, dynamic job can be created to model internal file system I/O transfer at any time during the simulation.

11.4 Limitations and Evolutions

The file system model described above has some limitations that are discuss here.

First, the storage model is very simple and do not reflect the fine grain behavior of different kind of storage like HDD, SSD, of NVM. Also, The storage model not include disk size limitation enforcement, and even if this can be done the decision process, the contention behavior that it implies is not modeled. To overcome the storage model limitation, it would be possible to add a more realistic model into SimGrid, but it may induce large changes in the underneath contention model.

The fact that the file system model is hold by the decision process increase flexibility and permits to the Batsim users to model any kind of behavior, but it has the drawback to lead to multiple implementation of the same model (one for each scheduler). This can be overcome by putting the file system model into an external process, which even more realistic, but it increase the maintenance cost and hinder the model genericity.

Another limitation is the absence of cache effects in the model, either from the storage itself or from a second level like burst buffers. Also, any cluster file system have metadata servers are also not taken into account in our model. To

allows to model fine grain behavior like the metadata server, or the cache effects, it requires to add new features inside Batsim, built on top of SimGrid directly, and thus putting the file system model inside Batsim.

CHAPTER 12

Interval set

Resources are represented as interval sets in the *Protocol* (e.g., in *EXECUTE_JOB*) and in some output files (e.g., *Jobs*). As the name suggests, an interval set is a set of intervals. More precisely, the intervals are closed integer intervals.

As an example, the set of resources $\{1, 2, 3, 5, 7\}$ equals to the $[1, 3] \cup [5, 5] \cup [7, 7]$ interval set.

12.1 Interval set string representation

Batsim uses a compact string representation of interval sets.

Each interval $[a, b]$ is represented as `a-b` or more simply by `a` if the degenerate case $a = b$. The delimiter between `a` and `b` is the minus sign (ASCII/UTF-8 0x2D). As an example, the $[37, 42] = \{37, 38, 39, 40, 41, 42\}$ interval is represented as `37-42`.

Interval sets are represented as intervals separated by a space (ASCII/UTF-8 0x20). As an example, the interval set $[1, 3] \cup [5, 5] \cup [7, 7] = \{1, 2, 3, 5, 7\}$ is represented as `1-3 5 7`.

Note: The same set of resources can have **many** string representations. For example, $\{1, 2, 3, 5, 7\}$ can be represented as `1-3 5 7`, `1-2 3 5 7`, `1 2 3 5 7`, `1-2 1-3 5 7` or even `5 2-2 7 1-3`.

12.2 Canonical string representation

The canonical string representation of an interval set respects the following rules.

- Intervals are disjoint (their intersection is empty).
- Intervals are as big as possible — *e.g.*, `1-3 4-5` is not a canonical representation of $[1, 5]$.
- Intervals of size 1 are represented as `a`, not `a-a`.
- Intervals are sorted in ascending order.

This representation is unique for an interval set and is the way to go if you need to manually generate interval set string representations.

All implementations should be able to read canonical string representations and to generate them. Canonical string representations are the only representations that are ensured to remain the same by being converted into an interval set and by being converted back to a string representation. $canon_repr \rightarrow intervalset \rightarrow canon_repr$.

12.3 Interval set libraries

Software libraries allow to operate interval sets from different programming languages.

- C++: intervalset (used by Batsim)
- D: dintervalset
- python: procset.py
- Rust: procset.rs

CHAPTER 13

Frequenty Asked Questions (FAQ)

13.1 Simulation / SimGrid

This section lists questions related to simulation or to SimGrid.

13.1.1 What is a parallel task?

Parallel tasks refer to two different SimGrid things.

- First, this is a high-level view of a set of computations and communications. As I write these lines, SimGrid proposes to use them via its parallel execution API `simgrid::s4u::this_actor::parallel_execute` (see SimGrid's documentation on executions on the CPU).
- Second, this is the SimGrid model `ptask_L07`, which is needed to use the SimGrid parallel execution API (the first item of this list). This model defines how computations are done on the CPU, but contrary to most other SimGrid CPU models `ptask_L07` also handles the progress of network activities at the same time.

A parallel task is a bundle of computations and network transfers that occur on a given number of hosts N. A parallel task is thus defined as:

- A computation amount (in number of floating-point operations) for each host (there are N values). Each value represent the amount of work executed by the host.
- A communication amount (in bytes) for each **directed** pair of host (there are N^2 values). Each value represent the amount of data sent from each host to another.

Note: A given parallel task is not bound to particular hosts, this is just a computation vector and a communication matrix.

An **ordered** list of hosts is given when you execute a parallel task, and hosts in the list do **not** need to be **unique**. This means you can instantiate a parallel task of size 4 in various ways:

- You can use hosts $\{1, 3, 5, 7\}$. The first computation value will be used on host 1, the second on host 3, the third on host 5, the fourth on host 7…

- You can use hosts {3, 1, 5, 7}. This is the same as before except that computation values will be placed differently: the first computation value will be used on host 3 and the second computation value will be used on host 1. **This change also affects network transfers** in a similar fashion.
- You can use hosts {1, 1, 1, 1}. In this case all computation will be done on the host 1, and all network transfers will be done on host 1 loopback link.

Related questions:

- *Can the scheduler decide the (inner) placement of a job?*
- *How to execute jobs that have per-core profiles?*

13.1.2 How is the execution of a parallel task simulated?

Contrary to a set of independent executions or network transfers, the sub-elements of a parallel tasks are **totally dependent**.

All sub-elements of a parallel task progress in a **completely synchronous** fashion. In other words, the advance rate of each individual computation and network transfer is **exactly** the same at each (simulation world) time.

Please note that this does not prevent parallel tasks progress to adapt to their execution context. The advance rate always remains the same for all sub-elements of a parallel tasks, but **this advance rate may have different values over (simulation world) time**. SimGrid computes what the current bottleneck is (the lowest advance rate among all sub-elements) at each simulation step, and apply this advance rate to all sub-elements of the parallel task.

For example, if a parallel task is executed alone (without any other computation or network transfer done on the simulated platform), it would progress at advance rate α_{alone}. If the same parallel task is executed while some of its resources (SimGrid hosts or links) are shared with other SimGrid activities (computations, network transfers, other parallel tasks…), it would progress at advance rate $\alpha \leq \alpha_{alone}$. Please note that the non-strictness of the inferior operator ($\leq$) is important here. If two parallel tasks share the same resources but the first parallel task's bottleneck is defined by executions, while the other parallel tasks's bottleneck is defined by network transfers, it is possible that the parallel task do not slow each other if they are run concurrently.

13.1.3 What is a CPU core in SimGrid?

A lie :).

Contrary to real-world CPU cores, SimGrid cores do not exist as distinct entities. In SimGrid, this is just properties of a host (hosts do exist as distinct entities in SimGrid). In particular, hosts have a per-core computation speed $core_{speed}$ (denoted as `speed` in SimGrid XML platforms as I write these lines) and a number of cores nb_{cores} (denoted as `core` in SimGrid XML platforms as I write these lines). These properties are used to limit the computation speed of activities on a host.

- The speed of each computation c is bound by the core computation speed: $speed_c \leq core_{speed}$.
- The sum of the speed of computations that occur on a host is bound by the core computation speed multiplied by the number of cores: $\sum_c speed_c \leq core_{speed} \times nb_{cores}$.

In short, this enables computations that occur on the same host to impact each other a lot when there are more computations than cores, while this impact does not exist if there are up to nb_{cores} computations (or is small depending on the CPU sharing model used).

Consequently, **Batsim does NOT enable schedulers to reserve or allocate cores** as they do not exist in SimGrid.

See also: *How to execute jobs that have per-core profiles?*.

13.1.4 Can the scheduler decide the (inner) placement of a job?

Yes. When you execute a job you essentially give the following information:

- A set of hosts `host_allocation` where the job shoud be executed. Please note that these allocations are currently given as the canonical representation of an *Interval set*, which **must be ordered**.
- A mapping that defines where each SimGrid executor should take place among `host_allocation`. This mapping is optional if you want one SimGrid executor per host (executor 0 will be put on host 0 of `host_allocation`, executor 1 will be put on host 1 of `host_allocation`…).

The number of SimGrid executors is defined by your simulation profile. Typically for parallel tasks, the number of SimGrid executors is the size of the computation vector.

If the simulation profile of your job is not homogeneous, the job inner placement (the mappingf is important as it may impact the job execution time.

13.2 How to execute jobs that have per-core profiles?

Currently, you have to use a custom execution mapping to do so (see *Can the scheduler decide the (inner) placement of a job?*).

For example if your profile is a parallel task of size 4 where each executor represents a core, you can decide to execute it on a single host (let us say host 5) by giving `host_allocation = 5` and a custom mapping that places all executors on the first allocated host `mapping = {"0": "0", "1": "0", "2": "0", "3": "0"}`. **This is possible regardless of the number of cores on host 5**, the number of cores on host 5 and their speed will only impact the job execution time (see *What is a CPU core in SimGrid?*).

You can decide another placement for that same parallel task of size 4. For example, you can decide to use two hosts and place the first two executors on the first allocated host (host 7), and the other two on the second allocated host (host 13). The following values describe this scenario.

- `host_allocation = 7 13`
- `mapping = {"0": "0", "1": "0", "2": "1", "3": "1"}`

Or to use the same two hosts, but with another placement strategy. Here, executors 0 and 2 will be on the first allocated host, while the others are on the second allocated host. The following values describe this scenario.

- `host_allocation = 7 13`
- `mapping = {"0": "0", "1": "1", "2": "0", "3": "1"}`

CHAPTER 14

Changelog

All notable changes to this project will be documented in this file. The format is based on Keep a Changelog. Starting with version *v1.0.0*, Batsim adheres to Semantic Versioning and its public API includes the following.

- The Batsim command-line interface.
- The format of the Batsim input files.
- The communication protocol with the decision-making component.

14.1 Unreleased

- Commits since v4.0.0
- `nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA batsim-master`

14.2 v4.0.0

- Commits since v3.1.0
- Release date: 2020-07-29
- `nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -i batsim-4.0.0`
- Recommended SimGrid release: 3.25.0 (see SimGrid's framagit releases)

14.2.1 Changed (breaks some schedulers)

- Profiles and jobs are now cleaned from memory over time (instead of at the end of the whole simulation). This is done with a reference counting mechanism: When a job or profile is no longer needed **according to what batsim knows**, it is removed from memory. This can break schedulers that rely on dynamic profile/job submission, especially when several *REGISTER_JOB* using the same profile are decided at different simulation times — as the profile can be garbage collected when its first execution finishes. The new `--enable-profile-reuse` *Command-line Interface* option should keep previous behavior.

14.2.2 Removed (breaks CLI)

- As unit tests are now done with gtest, the `--unittest` *Command-line Interface* option has been removed.

14.2.3 Added

- Scheduler configuration can be given to Batsim (via `--sched-cfg` or `--sched-cfg-file` *Command-line Interface* options). This configuration string is forwarded to the scheduler in the *SIMULATION_BEGINS* event.
- Basic tests for the external events mechanism.
- Retrieval of the zone properties in the XML platform description.
  - Platform properties declared within SimGrid zones are now retrieved and attached to each Batsim resource.
  - These properties are forwarded to the scheduler via the field `zone_properties` or each resource in the `compute_resources` and `storage_resources` arrays of the *SIMULATION_BEGINS* event.

14.2.4 Fixed

- Workflows crashed at the beginning and the end of the simulation. This should be fixed, and workflows are now tested under CI.
- Killing jobs should no longer issue memory issues (invalid reads and writes), which caused segmentation fault in corner cases — cf. issue 37 (inria).
- Killing sequences of delays should no longer crash with "Internal error" — cf. issue 108 (inria).
- SMPI profiles should now be automatically killed when their walltime is reached — cf. issue 95 (inria).

14.2.5 Miscellaneous

- Various performance improvements.
- The jobs output file is now written over time (was only written on disk at the end of the simulation).
- Batsim no longer uses SimGrid's MSG interface. Everything is done with S4U now.
- Smart pointers are used in most parts of the code (for reference counting memory deallocations).
- Old markdown documentation has been removed.
- Removal of CMake Find functions, pkgconfig is used instead.

14.3 v3.1.0

- Commits since v3.0.0
- Release date: 2019-05-26
- `nix-env -f https://github.com/oar-team/kapack/archive/master.tar.gz -i batsim-3.1.0`
- Recommended SimGrid release: 3.24.0 (see SimGrid's framagit releases)

14.3.1 Changed

- Batsim now requires that no *CALL_ME_LATER* are pending to send *SIMULATION_ENDS*.
- *Workload* identifiers are now generated depending on the order of the command-line arguments. Previously, they were hashes of the absolute filename of the workload, which was order independent.

14.3.2 Added

- A new *External Events* mechanism has been added.
  - For the moment the following external events are supported.
    - `machine_unavailable`: Some machines are no longer available.
    - `machine_available`: Some machines are available again.
    - *Generic_events*: User-defined external events that can be forwarded to the scheduler with the option `--forward-unknown-events`.
  - A new *NOTIFY* protocol event `no_more_external_event_to_occur` has been added to tell the scheduler that no more external events coming from Batsim can occur during the simulation.
  - A new command-line option was added: `--forward-unknown-events` that forwards unknown external events of the input files to the scheduler (ignored if there were no event inputs). The boolean value of this command is forwarded to the scheduler in the `SIMULATION_BEGINS` event.

14.3.3 Deprecated

- Building via CMake is deprecated. Next Batsim versions may only support Meson.

14.3.4 Miscellaneous

- Removed a build dependency to OpenSSL, which was only used to generate workload identifiers.
- Batsim integration tests are now written with pytest instead of CMake.

14.4 v3.0.0

- Commits since v2.0.0
- Release date: 2019-01-15
- `nix-env -f https://github.com/oar-team/kapack/archive/master.tar.gz -i batsim-3.0.0`
- Recommended SimGrid commit: 97b4fd8e4

14.4.1 Changed (breaks protocol)

- Removal of the `NOP` event.
- `SUBMIT_PROFILE` has been renamed *REGISTER_PROFILE*. Trying to register an already existing profile will now fail.
- `SUBMIT_JOB` has been renamed *REGISTER_JOB*. Trying to register an already existing job will now fail. The possibility to register profiles from within a *REGISTER_JOB* event has been discarded. Now use *REGISTER_PROFILE* then *REGISTER_JOB*.
- The *SIMULATION_BEGINS* event has been changed:
  - The `resources_data` array has been split into the `compute_resources` and `storage_resources` arrays.
  - The content of the `config` object has been flattened and now contains the following keys: `redis-enabled`, `redis-hostname`, `redis-port`, `redis-prefix`, `profiles-forwarded-on-submission`, `dynamic-jobs-enabled` and `dynamic-jobs-acknowledged`.
- The `submission_finished` *NOTIFY* event has been renamed `registration_finished`.
- The `continue_submission` *NOTIFY* event has been renamed `continue_registration`.

14.4.2 Changed (breaks command-line interface)

- Removal of the `--config-file` option. Everything should now be doable via the Batsim CLI.
- Removal of the `--enable-sg-process-tracing` option. You can now use `--sg-cfg` to do the same.
- `--batexec` has been renamed `--no-sched`.
- `--allow-time-sharing` has been split into two options `--enable-compute-sharing` and `--disable-storage-sharing`, as resource roles have been introduced.

14.4.3 Changed (breaks workload format)

- Profile types using parallel tasks have been renamed:
  - `msg_par` into `parallel` (see *Parallel task*)
  - `msg_par_hg` into `parallel_homogeneous` (see *Homogeneous parallel task*)
  - `msg_par_hg_tot` into `parallel_homogeneous_total` (see *Homogeneous parallel task with total amount*)

- msg_par_hg_pfs into parallel_homogeneous_pfs (see *Homogeneous parallel tasks with IO to/from a Parallel File System*)

14.4.4 Changed (breaks platform format)

- Batsim now uses SimGrid version 3.21 and therefore the SimGrid platform version 4.1, which broke things on how to define platforms. Please refer to SimGrid documentation for more information on this.

14.4.5 Changed (jobs/schedule output file format)

- **Breaks**: The columns requested_number_of_processors and allocated_processors have been respectively renamed requested_number_of_resources and allocated_resources in the jobs output file.
- **Breaks**: The order of the columns has changed in the jobs output file.
- The columns final_state and profile have been added in the jobs output file.
- The rejected jobs are now present in the jobs and the schedule output files.

14.4.6 Changed (new dependencies)

- docopt-cpp and pugixml are now external dependencies and no longer provided with Batsim sources.
- New intervalset dependency, which replaces the previous MachineRange class.
- batexpe is now an optional dependency to test batsim.

14.4.7 Added (protocol)

- Addition of the no_more_static_job_to_submit *NOTIFY* event, which is sent by Batsim when all the jobs described in the static workloads/workflows have been submitted.
- Addition of the profiles object in the *SIMULATION_BEGINS* event. The key is the workload_id and the value is the list of profiles of that workload.
- Addition of the optional storage_mapping object in the *EXECUTE_JOB* event, which allows to define which resource id should be used for a named IO resource.
- Addition of the optional additional_io_job object in the *EXECUTE_JOB* event, which allows to add IO movements to a job execution. This is done by merging a traditional parallel task (within the allocated hosts that *compute* the job) with another parallel task that define IO movements (within the allocated hosts that compute the jobs, but also potentially with IO resources).

14.4.8 Added (platform format)

- Roles can now be specified for the hosts of a platform. This is done by setting the role XML property of a host. A default master host can be specified this way by using the master role value. The storage value is for hosts that describe storage resources ; such hosts are allowed to send and receive bytes but not to compute. The compute_node value (used by default if no role is specified) is for hosts that describe computing resources that can both compute and communicate. More information in *Roles of hosts*.

14.4.9 Added (command-line interface)

- New `--add-role-to-hosts` option, that allows to add a role to some hosts.
- New `--sg-cfg` option, that allows to set SimGrid configuration options.
- New `--sg-log` option, that allows to set SimGrid logging options.
- New `--dump-execution-context` option, that dumps the command execution context on the standard output. This allows external tools to understand the execution context of a Batsim command without actually parsing it.

14.4.10 Known issues

- Killing jobs may now crash in some (corner-case) situations. This happens since Batsim upgraded its SimGrid version. Tracked on issue 37 (inria).
- SMPI profiles only handle relative trace filenames. Tracked on issue 97 (inria).
- Batsim does not check job size correctly when executed with `--no-sched`. Tracked on issue 70 (inria).

14.4.11 Miscellaneous

- Various bug fixes.
- Removed the python experiment scripts that were located in `tools/experiments`, as robin became the standard tool to execute Batsim experiments.
- Removed git submodules. Please now use schedulers directly from their repositories or from kapack.
- Removed dependencies to GMP and cppzmq.
- Batsim now mainly uses the s4u SimGrid interface. If you used to set SimGrid configuration/logging options through Batsim CLI, the name of such options should therefore have changed.
- Documentation moved to readthedocs.
- The `workload_profiles` directory has been renamed `workloads`.
- New generator for heteregenous platforms (code and documentation in `platforms/heterogeneous`).
- New demo (in `demo/`).

14.5 v2.0.0

- Commits since v1.4.0
- Release date: 2018-02-20
- `nix-env -f https://github.com/oar-team/kapack/archive/master.tar.gz -i batsim-2.0.0`
- Recommended SimGrid commit: 587483ebe

14.5.1 Changed (breaks protocol)

- The QUERY_REQUEST and QUERY_REPLY messages have been respectively renamed QUERY and ANSWER. This pair of messages is now bidirectional (Batsim can now ask information to the scheduler). Redis interactions with this pair of messages is no longer in the protocol (as it has never been implemented).
- When submitting dynamic jobs (SUBMIT_JOB), the job_id and id fields should now have the same value. Furthermore, jobs id are no longer integers but strings: my_wload!hello readers is now a valid job identifier.
- Removal of the job_status field from JOB_COMPLETED messages.
- JOB_COMPLETED messages should now be sent even for killed jobs. In this case, JOB_COMPLETED should be sent before JOB_KILLED.

14.5.2 Added

- Added the --simgrid-version command-line option to show which SimGrid is used by Batsim.
- Added the --unittest command-line option to run unit tests. Executed by Batsim's continuous integration system.
- New SET_JOB_METADATA protocol message, which allows to set set metadata to jobs. Such metadata is written in the _jobs.csv output file.
- The _schedule.csv output file now contains a batsim_version field.
- Added the estimate_waiting_time QUERY from Batsim to the scheduler.
- The *SIMULATION_BEGINS* message now contains information about workloads: A map from workload identifiers to their filenames.
- Added the job_alloc field to JOB_COMPLETED messages, which mentions which machines have been allocated to the finished job.

14.5.3 Changed

- The _jobs.csv output file is now written more cleanly. The order of the columns within it may have changed. Removal of the deprecated hacky_job_id field.

14.5.4 Fixed

- Numeric sort should now work as expected (this is now tested).
- Power tracing now works when the number of machines is big.
- Output buffers now work even if incoming texts are bigger than the buffer.
- The QUERY_REQUEST/QUERY_REPLY messages were not respecting the protocol definition (probably never tested since the JSON protocol update).
- Dynamically submitted jobs could not be used right away after being submitted (by the following events, or at least the events of the same timestamp). This should now be possible.

14.6 v1.4.0

- Commits since v1.3.0
- Release date: 2017-10-07
- `nix-env -f https://github.com/oar-team/kapack/archive/master.tar.gz -i batsim-1.4.0`
- Recommended SimGrid commit: 587483ebe

14.6.1 Added

- New `SUBMIT_PROFILE` protocol message that allows the decision process to submit profiles dynamically.
- New `msg_par_hg_tot` profile type. This is an homogeneous parallel task whose computation and communications amounts are spread over all allocated nodes. They can be seen as optimistic moldable tasks.

14.7 v1.3.0

- Commits since v1.2.0
- Release date: 2017-09-30

14.7.1 Added

- Jobs walltimes are no longer mandatory. The `walltime` field of jobs can now be omitted or set to -1. Such jobs will never be killed automatically by Batsim.

14.8 v1.2.0

- Commits since v1.1.0
- Release date: 2017-09-23

14.8.1 Added

- The job progress is now sent through the protocol when jobs are killed on request. This is done via a new `job_progress` map in `JOB_KILLED` messages, which gives this information for all the jobs that have really been killed.
- New job state `COMPLETED_WALLTIME_REACHED` (separated from `COMPLETED_FAILED`).

14.9 v1.1.0

- Commits since v1.0.0
- Release date: 2017-09-09

14.9.1 Added

- New job profiles `SCHEDULER_SEND` and `SCHEDULER_RECV` that communicate with the scheduler. New `send` and `recv` protocol events that correspond to them.
- Jobs now have a return code. Can be specified in the `ret` field of the jobs in their JSON description. Default value is 0 (success).
- New job state: `COMPLETED_FAILED`.
- New data added to the `JOB_COMPLETED` protocol event. `return_code` indicates whether the job has succeeded. The `FAILED` status can now be received.

14.9.2 Changed

- The `repeat` value of sequence (composed) profiles is now optional. Default value is 1 (executed once, no repeat).

14.10 v1.0.0

- Commits since v0.99
- Release date: 2017-09-09

14.10.1 Added

- Stated LGPL-3.0 license.
- Code cosmetics standards are now checked by Codacy.
- New PFS host. Associated with a new `hpst-host` command-line option.
- New protocol event `CHANGE_JOB_STATE`. It allows the scheduler to change the state of jobs in Batsim in-memory data structures.
- The `submission_finished` notification can be canceled with a `continue_submission` notification.
- New data to the *SIMULATION_BEGINS* protocol event. `allow_time_sharing` boolean is now forwarded. `resources_data` gives information on the resources. `hpst_host` and `lcst_host` give information about the parallel file system.
- New data to the `JOB_COMPLETED` protocol event. `job_state` contains the job state (as stored by Batsim). `kill_reason` contains why the job has been killed (if relevant).
- New `continue_submission` *NOTIFY* event, which cancels a previous `submission_finished` *NOTIFY* event.

14.10.2 Modified

- Improved and renamed parallel file system profiles.
- Improved code documentation.
- Improved the python scripts of the tools/ directory.
- Improved the python scripts of the test/ directory.

14.10.3 Fixed

- Complex allocation mapping were not handled correctly

14.11 v0.99

- Release date: 2017-05-26

14.11.1 Changed

- The protocol is based on ZeroMQ instead of Unix Domain Sockets.
- The protocol messages are now formatted in JSON (was custom text).

CHAPTER 15

Contact us

The main way to chat with the Batsim team is on Batsim's mattermost.

Otherwise, the batsim-user@inria.fr mailing list allows to ask questions and to get announcements. Registration is done on Inria's sympa.

CHAPTER 16

Platform

A Batsim platform is essentially a SimGrid platform. This page describes the specificities of Batsim platforms in comparison with SimGrid platforms. More information on how to define the SimGrid platform of your dreams can be found on the SimGrid documentation ;).

16.1 Roles of hosts

A host in Batsim can have one among several roles:

- `master`
- `compute_node`
- `storage`

In Batsim, the management of the simulation and the platform is given to a special host having the `master` role. There must and can be only one master host. The master host is a dummy host (it is not known by the scheduler) that is executing the server process and every other processes to manage the simulation, such as the request-reply process that talks to the scheduler, processes to submit launch and kill jobs, etc.

By default, every host of the platform, except for the `master host`, has the `compute_node` role, meaning that jobs performing computations can be executed on these hosts.

The last role a host can have is `storage`. A `storage` host should have a `speed` value of `0f`. Such host cannot perform any computation but can be used to execute jobs with IO profiles (more details about IO profiles can be found in the *Workload* documentation).

It is possible to specify the role of a host in the platform description by adding a property to that host. For example:

```
<host id="storage_server" speed="0f">
    <prop id="role" value="storage"/>
</host>
```

Roles can also be added via the *Command-line Interface* command `-r, --add-role-to-hosts <hosts_role_map>`, where `hosts_role_map` is formated as `<hostname1[,hostname2,...]>:<role>`.

16.2 Energy model

SimGrid allows each host to have a set of power states. Each state defines several properties.

- The computation speed (number of floating-point operations per second).
- The electrical energy consumption (in watts). This is usually defined with several values.

Switching from one power state to another is instantaneous in SimGrid — the idea is to allow simulators to implement whatever they want with the simple proposed model.

Batsim adds a simple layer on top of the SimGrid one so that switching from a power state to another may take a fixed amount of time and energy. This has been developed with node shutdown techniques in mind.

First, Batsim proposes to split the set of power states in three.

- Computation power states. These model energy and performance trade-offs of the computation machine — e.g., dynamic voltage and frequency scaling. **Only these can be used to compute jobs**.
- Sleep power states. These model how much energy is used during the sleeping state itself — e.g., node shutdown or suspend to RAM.
- Transition power states. These *virtual* power state model how long and how much energy takes the transition from one state to another.

Batsim users (= schedulers) can only switch into computation or sleep power states. Switching from one sleep power state to another is forbidden. Switching from one computation power state to another is instantaneous.

Todo:

- Finish to describe the Batsim energy model.
- Add instantiation examples.

CHAPTER 17

Workload

17.1 Overview and example

Workloads are one of the main Batsim inputs. They can be used to define what users desire to execute over time. Batsim separates workloads in two distinct sets that are **jobs** and **profiles**.

- Jobs define user requests. Typically, this is the information the scheduling algorithm can use to make its decisions.
- Profiles define what is inside applications. Typically, this is the information the platform simulator uses to simulate how the application should be executed.

Each job uses exactly one profile. Profiles can be shared by multiple jobs.

Workloads are defined in JSON. Here is an example of a Batsim workload from Batsim's repository (`workloads/test_various_profile_types.json`).

```
{
    "nb_res": 4,
    "jobs": [
        {"id":"delay", "subtime":0, "walltime": 30,  "res": 1, "profile": "delay"},
        {"id":"simple", "subtime":1, "walltime": 100, "res": 4, "profile": "simple"},
        {"id":"reach_walltime", "subtime":30, "walltime": 1,   "res": 4, "profile":
↪"simple"},
        {"id":"homo", "subtime":10, "walltime": 100, "res": 4, "profile": "homogeneous
↪"},
        {"id":"homo_no_cpu", "subtime":20, "walltime": 100, "res": 4, "profile":
↪"homogeneous_no_cpu"},
        {"id":"homo_no_com", "subtime":20, "walltime": 100, "res": 4, "profile":
↪"homogeneous_no_com"},
        {"id":"seq", "subtime":32, "walltime": 100, "res": 4, "profile": "sequence"},
        {"id":"2_resources", "subtime":15, "walltime": 30,  "res": 2, "profile":
↪"homogeneous_total"},
        {"id":"4_resources", "subtime":15, "walltime": 30,  "res": 4, "profile":
↪"homogeneous_total"}
```

(continues on next page)

(continued from previous page)

```
    ],

    "profiles": {
        "simple": {
            "type": "parallel",
            "cpu": [5e6,  0,  0,  0],
            "com": [5e6,  0,  0,  0,
                    5e6,5e6,  0,  0,
                    5e6,5e6,  0,  0,
                    5e6,5e6,5e6,  0]
        },
        "homogeneous": {
            "type": "parallel_homogeneous",
            "cpu": 10e6,
            "com": 1e6
        },
        "homogeneous_no_cpu": {
            "type": "parallel_homogeneous",
            "cpu": 0,
            "com": 1e6
        },
        "homogeneous_no_com": {
            "type": "parallel_homogeneous",
            "cpu": 2e5,
            "com": 0
        },
        "sequence": {
            "type": "composed",
            "repeat" : 4,
            "seq": ["simple","homogeneous","simple"]
        },
        "delay": {
            "type": "delay",
            "delay": 20.20
        },
        "homogeneous_total": {
            "type": "parallel_homogeneous_total",
            "cpu": 10e6,
            "com": 1e6
        }
    }
}
```

The following field must be defined in a workload file.

- `jobs` (array of jobs): See *Job definition*.
- `profiles` (object of profiles): See *Profile definition*.
- `nb_res` (positive integer): Indicates how many resources this workload has been designed for. Can be used to determine how many resources should be used in the simulation thanks to the `--mmax-workload` *Command-line Interface* argument.

Multiple input workloads can be given to Batsim (see *Command-line Interface*). While jobs and profiles are usually defined in workload files in a static manner, adding jobs and profiles dynamically (while the simulation runs) is possible. For more information about this, see *Dynamic registration of jobs*.

17.2 Job definition

Jobs must have the following fields.

- `id`: The job unique identifier (string).
- `subtime`: The job submission time (float, in seconds) — i.e., the **absolute** time at which the job request is issued in the system.
- `res`: The number of resources requested (positive integer).
- `profile`: The name of the profile associated with the job (string) — i.e., the definition of how the job execution should be simulated.

Some optional fields are used by Batsim.

- `walltime`: By default, jobs have no execution time limit. Setting a value (float, in seconds) to the `walltime` field makes Batsim automatically stop a job that exceeds its walltime. In constrast to `subtime`, this value is **relative** to the start of the job.

Users can define any other field as they desire. Such information is not directly used by Batsim but is forwarded to the scheduler at the job submission time. The scheduler can then use the additional information. Here is a **non-exhaustive** list of what this workload definition flexibility allows.

- Defining dependencies between jobs. This can be done by adding a `dependencies` field in jobs, which is a list of other job names.
- Constraining where jobs can be executed. For example, users may desire that their jobs are executed as locally as the platforms allows it (e.g., with a `must_be_local` boolean). Some other users may ask for a specific set of machines. Some other users may want to indicate that the job can only be used on a special kind of machines (e.g., on GPGPUs)…
- Adding shared resources to the job that are not machines. This is for example the case for proprietary software licenses: The number of concurrent MATLAB executions on a platform may be limited.
- Specifying which queue the job comes from. Please note that giving multiple workloads to Batsim is also possible (see *Command-line Interface*).
- Adding metainformation about how the job has been generated. This can be helpful if one wants to assess advanced workload generation techniques.

17.3 Profile definition

Profiles must have the following fields.

- `type`: The type of profile (string). If the profile is executed by Batsim, Batsim must know the profile type. See *Profile types overview* for an overview of the profile types whose execution is directly supported by Batsim.

Some optional fields are used by Batsim.

- `ret`: The profile execution return code (integer) in case of success (execution finished *normally*, i.e., without reaching the job's walltime and without being killed by the scheduler). Default value is 0, meaning success. Overriding this value can be useful if the job is supposed to fail even when it finishes *normally*, for example to model that the user gave an invalid application execution script. Non-zero return codes will translate into non-success in the *Jobs* output file.

Other fields may be used by Batsim depending of the profile type.

Users can define any other field as they desire. This is less useful than for jobs, as profiles are usually not forwarded to the scheduler (but it can be enabled, see *Command-line Interface*). It has however proved to be convenient in some situations. For example, we defined workloads that can be executed both in simulation and on a real distributed systems via OAR in Batsim's initial article thanks to an additional `command` field to define how each job should be executed on the real system.

17.4 Profile types overview

Here are listed the main types of profiles understood by Batsim — in addition to examples for each profile type.

17.4.1 Delay

This is the simplest profile type. In fact there is no job execution but only a fixed number of seconds during which the machines will sleep.

It does **not** take the platform into account at all. It cannot be used to see any network or CPU contention. It cannot be used directly to observe the energy used by the job — it would be similar to remaining idle.

The following example defines a profile that always waits for 20.20 seconds.

```
{
  "type": "delay",
  "delay": 20.20
}
```

Note: In fact, a job execution with the previous delay can be faster than 20.20 seconds if the job's walltime is smaller that 20.20.

17.4.2 Parallel task

This profile type defines a set of computations and communications whose execution is tightly bound. In other words, at any given time during the profile execution, the progress rate of every communication and computation will be the same.

Parameters.

- `cpu`: An array defining the amount of floating-point operations that should be computed on each allocated machine.
- `com`: An array defining the amount of bytes that should be transferred between allocated machines. This is in fact a matrix where host in row sends to host in column. When row equals column, the communication is done through the machine loopback interface (if defined in the *Platform*).

Here is an example of a parallel task that can be used by any job requesting 4 machines.

```
{
  "type": "parallel",
  "cpu": [5e6,  0,  0,  0],
  "com": [5e6,  0,  0,  0,
          5e6,5e6,  0,  0,
```

(continues on next page)

(continued from previous page)

```
          5e6,5e6,  0,  0,
          5e6,5e6,5e6,  0]
}
```

The first allocated machine of such a profile will compute $5x10^6$ floating-point operations, while the other machines will not compute any floating-point operation. The picture below illustrates the communications done within the parallel task. All allocated machines will send $5x10^6$ bytes to the first allocated machine. The second allocated machine will send $5x10^6$ bytes to the first and second allocated machines…

The execution of such profiles is context-dependent. The computing speed of the machines and the network properties (essentially the bandwidth) is directly taken into account by SimGrid to compute the job execution time.

This profile type allows to observe large-grained interference phenomena between jobs, involving shared computing machines and the bandwidth of shared network nodes. It can be used to model applications whose execution is very smooth. Please note that it is probably not realistic enough to observe fine-grained phenomena, such as the impact of network latency when the application heavily relies on short messages that limit its control flow. If you are in such a case, the *SMPI trace replay* profile type may interest you.

17.4.3 Homogeneous parallel task

This profile type is a convenient way to generate an homogeneous *Parallel task* that can be used by any job, regardless of the number of machines it requests.

Parameters.

- `cpu`: The amount of floating-point operations that should be computed on each machine.
- `com`: The amount of bytes to send and receive between each pair of distinct machines. The loopback communication of each machine is set to 0.

```
{
  "type": "parallel_homogeneous",
  "cpu": 10e6,
  "com": 1e6
}
```

17.4.4 Homogeneous parallel task with total amount

This profile type is a convenient way to generate an homogeneous *Parallel Task* by giving the total amount of work to be done. It allows such profiles to be used with any number of resources while conserving the same amount of work to do.

Note: This can help modeling moldable jobs with the help of *Dynamic registration of jobs*.

Parameters.

- `cpu`: The total amount of floating-point operations that should be computed over all nodes. Each node will have an amount of $cpu/node_count$ floating-point operations to compute, where $node_count$ is the number of nodes allocated to the job.
- `com`: The amount of bytes that should be sent and received on each pair of distinct nodes. Each node will send and receive an amount of $com/node_count$ bytes. The loopback communication of each node is set to 0.

```
{
  "type": "parallel_homogeneous_total",
  "cpu": 10e6,
  "com": 1e6
}
```

17.4.5 Sequence of profiles

This profile type defines a list of other profiles that should be executed in sequence.

Parameters.

- `seq`: The array of profile names that should be executed.
- `repeat` (optional): The number of times the sequence will be repeated. By default, the sequence is only executed once (value is 1).

```
{
  "type": "composed",
  "repeat" : 4,
  "seq": ["prof1","prof2","prof1"]
}
```

17.4.6 Homogeneous parallel tasks with IO to/from a Parallel File System

Represents an IO transfer between all the nodes of a job's allocation and a centralized storage tier. The storage tier is represented by one host of the *Platform* having the role `storage`.

Parameters.

- `bytes_to_read`: The amount of bytes to read from the PFS to each node (float).
- `bytes_to_write`: The amount of bytes to write to the PFS from each node (float).
- `storage`: (optional) A label for the storage to use (string). It will be mapped to a specific node at the job execution time. Default value is `pfs`. See *EXECUTE_JOB* for more details about the `storage_mapping`.

```
{
  "type": "parallel_homogeneous_pfs",
  "bytes_to_read": 10e5,
  "bytes_to_write": 10e5,
  "storage": "nfs"
}
```

17.4.7 Staging parallel tasks between two storage tiers

This profile type represents an IO transfer between two storage tiers. Storage tiers are hosts of the *Platform* having the role `storage`.

Parameters.

- `nb_bytes`: The amount of bytes to be transferred (float).
- `from`: A label for the sending storage tier (string). It will be mapped to a specific host at the job execution time.
- `to`: A label for the receiving storage tier (string). It will be mapped to a specific host at the job execution time.

See *EXECUTE_JOB* for more details on the `storage_mapping` needed for both the `from` and the `to` fields.

```
{
  "type": "data_staging",
  "nb_bytes": 10e5,
  "from": "pfs",
  "to": "nfs"
}
```

17.4.8 SMPI trace replay

Profiles of this type correspond to the replay of a SMPI time-independent trace. Such traces allow to see the fine-grained behavior of MPI applications.

Note: This profile type may not be realistic with all applications, as the application is simulated *offline*: The application is first executed to get a trace, then the trace is replayed.

This may be wrong if the application logic depends on the execution context, for example if the application communication pattern depends on the observed latencies at runtime.

Parameters.

- `trace`: The file name of the main trace file (string).

Warning: As I write these lines, the `trace` filename must be relative to the Batsim workload file in which the profile is defined.

As a full example, refer to the trace in `workloads/smpi/compute_only` and to the `workloads/test_smpi_compute_only.json` workload file.

Warning: As I write these lines, walltime is not implemented for `smpi` jobs.

```
{
  "type": "smpi",
  "trace": "smpi/compute_only/traces.txt"
}
```

CHAPTER 18

External Events

18.1 Overview and example

This is a mechanism to enable injection of external events during the simulation. For example, one would be interested in studying the behavior and resilience of a scheduling policy when machine can become unavailable for a certain period of time, which is made possible using the events injection mechanism!

18.2 Input File Format

An example of an input file for external events can be found in `events/test_events_4hosts.txt`. Such file should contain on each line a JSON description of an event, as in

```
{"type": "machine_unavailable", "resources": "0", "timestamp" : 0}
{"type": "machine_available", "resources": "0", "timestamp" : 10}
{"type": "machine_unavailable", "resources": "0-1 3", "timestamp" : 20}
{"type": "machine_available", "resources": "0 1", "timestamp" : 250}
{"type": "machine_available", "resources": "3", "timestamp" : 300}
```

Note that it is not mandatory to have the events ordered by their timestamps in the input files, as they will be sorted anyway once all events of a file are loaded by Batsim. In each event description, the field `type` contains the type of the external event that occurs and the field `timestamp` contains the date at which the event has to occur during the simulation. Other fields may be present, depending on the event type, as described in *Supported Events*.

18.3 Supported Events

Here is the list of supported external events that can be injected during the simulation:

- *Machine_unavailable*
- *Machine_available*

Each external event is notified to the scheduler via a *NOTIFY* protocol event.

18.3.1 Machine_unavailable

The machines specified by the *resources* field (represented as an *Interval set*) become unavailable. It is no longer possible to execute jobs on these machines, but **jobs that were already running on these machines are not killed**. To kill such jobs, please make your decision process send a *KILL_JOB* event on notification of machines becoming unavailable.

```
{"type": "machine_unavailable", "resources": "0-4 6 8", "timestamp" : 10}
```

18.3.2 Machine_available

The machines specified by the *resources* field (represented as an *Interval set*) become available. It is now possible to execute jobs on these machines **if no job is running on them** (unless resource sharing is enabled).

```
{"type": "machine_available", "resources": "2-4 8", "timestamp" : 20}
```

18.3.3 Generic_events

External events unknown to Batsim can also be added in the input file. These events are not supported by Batsim unless the option `--forward-unknown-event` is set. In this case, such events will be forwarded to the scheduler without any check (except for the *type* and *timestamp* fields).

```
{"type": "user_specific_type", "timestamp": 150, "some_field": "value_of_field",
↪"another_field": 36}
```

CHAPTER 19

Schedule

Aggregated information about the schedule is exported as *prefix* + `_schedule.csv` (see *Command-line Interface*). The file is formatted as CSV. It only contains a header and a single entry for the whole simulation. This file contains the following fields in lexicographic order of the fields name.

- `batsim_version`: Similar to the output of the `--version` *Command-line Interface* option.
- `consumed_joules`: The total amount of joules consumed by the machines from the submission time of the first job to the finish time of the last job.
- `makespan`: The completion time of the last job.
- `max_slowdown`: The maximum slowdown observed on a job. Slowdown is computed for a job as its turnaround time divided by its execution time.
- `max_turnaround_time`: The maximum turnaround time observed on a job. Turnaround time is computed for a job as its completion time minus its submission time.
- `max_waiting_time`: The maximum waiting time observed on a job. Waiting time is computed for a job as its starting time minus its submission time.
- `mean_slowdown`: The average slowdown observed on jobs. Slowdown is computed for a job as its turnaround time divided by its execution time.
- `mean_turnaround_time`: The average turnaround time observed on jobs. Turnaround time is computed for a job as its completion time minus its submission time.
- `mean_waiting_time`: The average waiting time observed on jobs. Waiting time is computed for a job as its starting time minus its submission time.
- `nb_computing_machines`: The number of computing machines in the simulation.
- `nb_grouped_switches`: The number of host power state transitions requested by the decision process.
- `nb_jobs`: The number of jobs in the simulation.
- `nb_jobs_finished`: The number of finished jobs in the simulation.
- `nb_jobs_killed`: The number of killed jobs in the simulation.
- `nb_jobs_success`: The number of jobs that finished successfully in the simulation.

- `nb_machine_switches`: The number of host power state transitions done on machines. This can be seen as a *flattened* version of `nb_grouped_switches` over machines.
- `scheduling_time`: The (real world) time (in seconds) spent in the scheduler (and in the network).
- `simulation_time`: The (real world) duration (in seconds) of the whole simulation.
- `success_rate`: $nb_jobs_success/nb_jobs$
- `time_computing`: Total time of all machines spent in computing state.
- `time_idle`: Total time of all machines spent in idle state.
- `time_sleeping`: Total time of all machines spent in sleeping state.
- `time_switching_off`: Total time of all machines spent in switching_off state.
- `time_switching_on`: Total time of all machines spent in switching_on state.

CHAPTER 20

Jobs

The main Batsim output file is exported as *prefix* + `_jobs.csv` (see *Command-line Interface*).

This file is formatted as CSV (with a header) and give information about jobs. There is one line per job. This file has the following fields in this order.

- `job_id`, the job identifier. Value is unique within a workload.
- `workload_name`, the name of the workload the job belongs to.
- `profile`, the name of the profiles that defines how the job should be simulated.
- `submission_time`, the (simulation world) time (in seconds) at which the job has been submitted.
- `requested_number_of_resources`, the number of resources requested by the job.
- `requested_time`, the requested time of the job (walltime).
- `success`, whether the job has completed before reaching its walltime and returned 0.
- `final_state`, the job final state. Possible values include the following.
  - `COMPLETED_SUCCESSFULLY`: Job executed without reaching walltime, with zero exit code.
  - `COMPLETED_FAILED`: Job executed without reaching walltime, with non-zero exit code.
  - `COMPLETED_WALLTIME_REACHED`: Job executed but reached its walltime (killed automatically).
  - `COMPLETED_KILLED`: Job executed but killed by the decision process.
  - `REJECTED`: Job not executed, it was rejected by the decision process.
- `starting_time`, the (simulation world) time (in seconds) at which the job execution has been started.
- `execution_time`, the (simulation world) duration (in seconds) of the job execution. Equals to $finish_time - starting_time$.
- `finish_time`, the (simulation world) time (in seconds) at which the job execution has finished.
- `waiting_time`, the (simulation world) time (in seconds) the job waited before being executed. Equals to $starting_time - submission_time$.
- `turnaround_time`, the time the job spend in the system. Equals to $finish_time - submission_time$.

- `stretch`: equals to $turnaround_time/execution_time$.
- `consumed_energy`, the total amount of energy (in joules) consumed by the `allocated_resources` during the execution of the job. Warning: no energy sharing or allocation is done in case there is more than one job running on a machine.
- `allocated_resources`, the resources onto which the job has been allocated (see *Interval set string representation*).
- `metadata`, user-specified metadata about the job (empty string by default).

Please note that many fields can have empty values for jobs that have been rejected.

CHAPTER 21

Energy

When a Batsim simulation is run with energy enabled (see *Command-line Interface*), Batsim outputs energy-specific output files.

You can give a look at this evalys visualization example to see how to use such data.

21.1 Energy consumption trace

This file is written as Batsim's export prefix + `_consumed_energy.csv`. This is a time series that contains the energy consumption of the platform (as defined as the sum of all the computing hosts of the platform) from time 0 to the current time. It contains the following fields in this order.

- `time`: The time point at which the measure has been done.
- `energy`: The amount of energy consumed in joules by the platform from time 0 to `time`. **This metrics is robust, you can analyze/visualize it without concerns.**
- `wattmin`: The minimum current power consumption of the platform, taking into account the power state into which each host is, and assuming that all hosts have an *epsilon* load (close to 0 without reaching it).
- `epower`: The average power consumption of the whole platform since last event to the current one. **Use this value with caution** as it can be subject to degenerate cases — *e.g.*, when two successive events happen at the same `time`.
- `event_type`: The type of the event that occurred at time `time`.
  - `s` if the event is a job start
  - `e` if the event is a job end
  - `p` if the event is a host power state change

21.2 Power state change trace

This file is written as Batsim's export prefix + `_pstate_changes.csv`. This is a time series that contains the power state transitions of the hosts over time. It contains the following fields in this order.

- `time`: The time at which the power state transition occurred.
- `machine_id`: The *Interval set* of hosts whose power state has been changed.
- `new_pstate`: The new power state (an integer) of the involved hosts.

21.3 Agregated machine state trace

This file is written as Batsim's export prefix + `_machine_states.csv`. This is a time series that contains the number of hosts in each kind of power state over time. Please refer to *Energy model* for more information about the existing kinds of power states. It contains the following fields in this order.

- `time`: The time at which the measure has been done.
- `nb_sleeping`: The number of hosts currently in a sleeping power state.
- `nb_switching_on`: The number of hosts currently transitioning into a computation power state.
- `nb_switching_off`: The number of hosts currently transitioning into a sleeping power state.
- `nb_idle`: The number of hosts currently in a computation power state, but without a job running on them.
- `nb_computing`: The number of hosts currently in a computation power state, with a job running on them.

CHAPTER 22

Contributing guidelines

- Please do *Contact us* in case of doubt.
- You found a bug or desire a feature? *Open an issue*.
- You want to improve Batsim's code or documentation? Read *Setting up a development environment* then *Open a merge/pull request*.

22.1 Open an issue

This is mainly done on Inria Gitlab issues, but also on GitHub issues for external users.

If you found a bug, please provide a minimal working example so we can reproduce it. Please also tell us about your software environment. Giving versions for Batsim, SimGrid and the scheduler you use is especially important.

22.2 Open a merge/pull request

This is mainly done on Inria Gitlab MRs, but also on GitHub PRs for external users.

Do not hesitate to *Contact us* or to *Open an issue* before implementing your work. This can save you some time, especially if the contribution does not match *Batsim objectives*.

If you plan to do several unrelated improvements, please do several merge/pull requests. Furthermore, please respect the following git usage.

- Create a dedicated git branch for your bugfix or feature implementation.
- Base your branch on Batsim's up-to-date master branch.
- Do not put unrelated commits in the branch.
- Do not put merge commits in the branch.

Note: If you already have modifications and do not know how to make them fit this usage, please read Atlassian's git tutorial on rebasing.

22.3 Batsim objectives

- Reduce maintenance cost. As we lack development manpower, this is very important.
- Keep it simple, stupid. We try to avoid very specific code in Batsim itself. Ideally, the features should be modular enough to allow a wide use case variety. For example, the *Protocol* should propose a set of basic operations that can be composed to achieve what you want, rather than a dedicated operation that does exactly what you want.
- Performance should be kept in mind, even if this is not the main concern.

CHAPTER 23

Setting up a development environment

Nix is used heavily in the Batsim ecosystem. Not only to distribute the software to end users, but also to manage the build and test environments of the various tools. In the end, we recommend Nix to any Batsim user or developer:

- For end users, it provides controlled environments to execute your simulations, which helps a lot in making your experiments repeatable (yes, you/your advisors/your reviewers/etc. want them to be repeatable :p). It also avoids the hassle of installing dependencies. Please give a look at *Using Batsim from a well-defined Nix environment* for a simple Nix usage in this case, or follow our *Doing a reproducible experiment* tutorial to see more advanced usage (tuning/pinning the versions of the various tools).
- For developers getting started with Batsim, we provide environments to build or test Batsim, which means no hassle to install dependencies regardless of your Linux distribution or operating system. It also means that you can trivially share your development environment, which will enable us to help you more easily if you *Contact us*.
- For advanced Batsim developers, Nix's full power can be unleashed to tune various packages at the same time while remaining clean/reproducible/shareable with other developers. This is especially useful when adding a new Batsim feature that involves modifying Batsim and schedulers at the same time — or when you also need to modify SimGrid for your work.

23.1 Nix architecture

Most of our packages are defined in the NUR-Kapack repository. The default versions of packages defined in NUR-Kapack are meant to be used for end users: Optimized binaries without debug information.

The daily environment used to build and test Batsim (by developers or *Continuous Integration* machines) is different, as in this case we want debug information on Batsim but also on its dependencies, notably SimGrid. All of this environment is defined in `default.nix`, located at the root of Batsim's git repository. This file reuses the package definitions from NUR-Kapack and *overrides* them to have the desired behavior. This file defines a Nix set, that is to say an associative structure that here associates names (*attributes*) to packages or environments (Nix derivations). Here is an overview of the various attributes of this file:

- `batsim` defines a package that builds the Batsim executable (and coverage information generated by gcov at compile time). This package may also run unit tests and generate coverage information about them.

- `integration_tests` defines a package that runs Batsim integration tests and generates test report and coverage information.
- `coverage-report` combines gcov information to generate readable coverage reports.
- `sphinx_doc` generates Batsim's sphinx documentation (you are currently reading it).
- `doxydoc` generates Batsim's code doxygen documentation.

23.2 Basic usage with `nix-build`

Most of our *attributes* are meant to be used easily with `nix-build`. As the name says, `nix-build` builds a Nix expression and puts the result files in the Nix store — a `./result` symlink is also produced so you can easily give a look at the files generated by the command.

You can for example build a debug version of Batsim from local sources with `nix-build -A batsim`, then run it with `./result/bin/batsim` (do not worry if you see many warning lines about profiling/gcda files, this is a normal artifact of our gcov usage).

Note: The first time you call this command, Nix may compile many dependencies if they are not present in your machine's cache. Most dependencies are likely to be present in Batsim's binary cache, so you can simply download them from there if you want to save some time — refer to *Using Nix* to enable fetching data from Batsim's binary cache with `cachix`.

You can then run integration tests with `nix-build -A integration_tests`. Please note that Nix will compile/download any needed dependency when this command is called. Here, this means that Batsim (from the `batsim` attribute) will be compiled if needed, or schedulers (`batsched` and `pybatsim` attributes), or tools to run the tests (`pytest` and `batexpe`).

Finally, you can see which Batsim code lines are covered by our tests by calling `nix-build -A coverage-report`. By default, this should generate textual reports (plain text, CSV and JSON) — `cat ./result/file-summary.txt` prints one of the generated files. You can enable many other output formats by setting Nix variables when calling commands. For example, `nix-build --arg coverageHtml true -A coverage-report` will generate human readable html files where you can browse lines of code and see whether they are covered or not with `firefox ./result/html/index.html`. For a list of available arguments, please read the first lines of `default.nix`.

Warning: By default, the `integration_tests` attribute will always be rebuilt (when you call `nix-build -A integration_tests` directly or when you use any attribute that requires it such as `nix-build -A coverage-report`). This is done voluntarily as some tests may have non-deterministic outcome and we want to easily run them many times in a row.

If for some reason you want to avoid rebuilds (typically if you want to generate coverage reports from an existing test run), you can define the `testVersion` Nix variable to a fixed value when running your commands: `nix-build --argstr testVersion fixed -A integration_tests` then `nix-build --argstr testVersion fixed -A coverage-report`. You can put any value instead of `fixed` as long as you use the same value in all your commands.

23.3 Iterative builds with `nix-shell`

`nix-build -A batsim` makes sure that Batsim is built correctly by building Batsim sources from scratch every time. This is an interesting property, but sometimes you just want to compile many times in rapid succession, and keeping/updating a build cache to compile Batsim can be very useful. The simplest way to achieve this is to get into an environment that can build Batsim, and to compile it manually from there.

- Enter shell that can build Batsim: `nix-shell -A batsim`
- From inside the shell, generate a Meson build directory if it does not already exist: `meson build`
- From inside the shell, compile Batsim: `ninja -C build`

You should now be able to edit your code however you want then call `ninja -C build` to trigger a Batsim build with cache.

> **Warning:** Please note that you may need to recreate the build directory depending on what you do. Typically, whenever you add new files or change the include graph of the project, creating a new build directory is advised.

23.4 Using IDEs such as `qtcreator`

Depending on what development you are doing, using an IDE such as `qtcreator` can be useful to edit source code. This is completely doable, but you have to make sure to run your IDE from the right environment. Assuming that `qtcreator` is already installed in your local environment, you can follow these steps to make it able to build Batsim.

- Enter a shell that can build Batsim with CMake: `nix-shell -A batsim_cmake`
- From inside the shell, generate a CMake build directory if it does not already exist: `(mkdir build-cmake && cd build-cmake && cmake .. && make -j $(nproc))`
- From inside the shell, run qtcreator to import an existing CMake project: `qtcreator ./CMakeLists.txt &`. When qtcreator prompts you about how to import the project, tell it to only use the existing build environment/directory and **not** to create a new one.
- You should now be able to compile Batsim from qtcreator.

23.5 Run tests from iteratively-built Batsim

Sometimes, you want to run some Batsim integration tests from an iteratively-built Batsim. Typically, this happens when you are making a single test pass and you want to do fast build/test iterations. This can be done by using several environments at the same time:

- Keep a first environment open, in which you build Batsim. These environments can be *Iterative builds with nix-shell* or *Using IDEs such as qtcreator* for example.
- Keep a second *hacky* environment open, in which you run the tests.

You can create the second environment by calling `nix-shell -A integration_tests` — **the following commands of this section are to be executed in the shell created by this nix-shell command**. The default `batsim` executable in this environment is the one built cleanly by Nix. You can call `which batsim` to see where the `batsim` executable is located (spoiler: in the Nix store). As Batsim integration tests simply run Batsim by calling a `batsim` command, **you can hack the test environment to run your iteratively built Batsim by hacking your PATH environment variable**. For example, if your Batsim build cache directory is in `${HOME}/proj/batsim/build`, you

can hack your call path via this command: `export PATH="${HOME}/proj/batsim/build:${PATH}"`. To make sure the hack worked, call `which batsim` to make sure that `batsim` now references your iteratively built file.

Now, in the second shell, you can go into the `test` directory (from Batsim's root directory) and run tests how you want (`pytest` will run all tests, `pytest test_energy.py` will run energy tests. . .).

23.6 Running a simulation instance with gdb

Sometimes, you want to run Batsim/the scheduler/both with a debugger to see what's going on — typically when a test fails and you modify Batsim's source code in a trial and error method. This section shows how this can be done (we will only run Batsim with a debugger here).

First, please read *Run tests from iteratively-built Batsim*, as this section is a direct continuation to it. Here, we will need three environments:

- A first environment to build Batsim iteratively. Make sure to build Batsim without optimization and with debug information (DWARF symbols). This should be done by default by Meson but you can force it if needed, refer to Running Meson and Meson's built-in options in this case. Calling `file ./build/batsim` on your iteratively built batsim should show whether debug information is present or not (`with debug_info, not stripped`).
- A second environment to execute Batsim. A starting point of this environment is the one defined in *Run tests from iteratively-built Batsim*. **Unless stated explicitly, all commands listed in the following of this section are to be run in this environment**.
- A third environment to execute the scheduler. As here we won't modify the scheduler, just use `nix-shell -A integration_tests`.

The main entry point of Batsim's integration tests is to call pytest, that will generate many simulation instances and run them. Most simulation instances consist in calling `batsim` with some command-line arguments, and a scheduler (typically `batsched`, sometimes `pybatsim`) with some command-line arguments. A wrapper process called `robin` is in charge of executing both commands and of handling errors (typically, kill batsim when scheduler crashes and vice versa, or kill everyone when a timeout is reached to avoid infinite loops). When `robin` runs a simulation instance, it generates separate scripts to run Batsim and the scheduler. Here, we will see how these scripts can be hacked to run Batsim from gdb.

First, clean the output directory of Batsim's tests, which is located in `test/test-out` from Batsim's source directory. Then, run integration tests with `pytest` from the `test` directory — here we will assume that one of the tests defined in `test_walltime.py` fails so we will run `pytest test_walltime.py`. Running this command will populate the `test-out` directory with many tests: A directory is created per simulation instance, and an `instance.yaml` file describes how to run each instance. You can manually run a simulation instance by calling `robin instance.yaml` (assuming your have moved your current working directory to the test you want to work on). You can run several tests until you find the one that fails. Now, assuming you found the failing test you want to work on, you can give a look at the test directory structure — `tree` should output something like this:

```
.
├── batres_jobs.csv
├── batres_machine_states.csv
├── batres_schedule.csv
├── batres_schedule.trace
├── cmd
│   ├── batsim.bash
│   └── sched.bash
├── instance.yaml
└── log
```

(continues on next page)

(continued from previous page)

```
    ├── batsim.log
    ├── sched.err.log
    └── sched.out.log
```

- Files that start with batres_ are simulation results generated by Batsim.
- The log directory contains the logs of the batsim/scheduler commands.
- The cmd directory contains the commands that were run by robin.

You can now run the simulation instance without robin, by calling batsim and the scheduler in their own shells.

- In the shell configured to run your iteratively built Batsim, you can run ./cmd/batsim.bash.
- In the shell for the scheduler, you can run ./cmd/sched.bash.

Now you just have to hack ./cmd/batsim.bash so that it runs batsim with your favorite debugger, and execute both scripts in their own shells again. As I personally use gdb with the cgdb terminal interface, so I just prefix the Batsim command with cgdb --args to run it inside my debugger.

Note: Your debugger may not display some source files. This should not happen for Batsim source code, but it may happen for dependencies such as SimGrid, in which case you can observe something like this:

```
(gdb)
#8  0x00007ffff7a1270c in simgrid::kernel::context::Context::operator()␣
↪(this=0x7f9d00) at ../src/kernel/context/Context.hpp:65
65      ../src/kernel/context/Context.hpp: No such file or directory.
```

If you want to display SimGrid source code, clone SimGrid's source code somewhere in your filesystem (make sure to checkout the exact same version as the one defined in default.nix/NUR-Kapack) and tell your debugger where to find source files. If you cloned SimGrid in /home/user/proj/simgrid-release and you use gdb, this is done by typing dir /home/user/proj/simgrid-release/src in gdb's interactive interface.

You can do the exact same trick for any source code that your debugger cannot find. You can also create initialization files for your debugger to load these directories automatically.

23.7 Changing Batsim dependencies

Sometimes, you not only need to hack Batsim but also one of its dependencies, which happens quite often with SimGrid. In this section we will see how default.nix can be hacked to use a custom SimGrid version, that can either come from your own git repository (fork) or just from your local filesystem.

As simgrid is an *input* of default.nix, its value can be overridden when default.nix is evaluated by giving command-line arguments to nix-build or nix-shell without modifying the file at all. However, we will **not** take this approach here as modifying default.nix is easier to do. We will instead create a new SimGrid package called my-custom-simgrid then tell the batsim package to use our new custom SimGrid. As I write these lines, default.nix defines and returns a Nix set named jobs. It's **inside** jobs that we will add our new package. Here is a first version of my-custom-simgrid

```
my-custom-simgrid = (kapack.simgrid-light.override { inherit debug; }).overrideAttrs␣
↪(attr: rec {
  src = pkgs.fetchgit {
    rev = "44bca631e482bd6e48a926393437d0959b661218";
    url = "https://framagit.org/mpoquet/simgrid.git";
```

(continues on next page)

(continued from previous page)

```
    sha256 = "sha256:1fs0sdwx77yrakbffh00g7hxqladbjpqcs15lcx37viahjlk7fp0";
  };
});
```

Here is an explanation of the various subparts of this Nix expression:

- `my-custom-simgrid = ...;` defines a new *attribute* named `my-custom-simgrid` in the embracing set (named `jobs`).
- `(kapack.simgrid-light.override { inherit debug; })` *overrides* the **inputs** of the `simgrid-light` SimGrid version defined in NUR-Kapack, by customizing its `debug` input. `inherit debug;` is strictly equivalent to `debug=debug;`. Most packages in NUR-Kapack have a `debug` input that make packages generate and keep debug information (DWARF symbols).
- `<package>.overrideAttrs (attr: rec {...})` *overrides* the **definition** of `<package>` (here `package` is a lightweight SimGrid with debug information). This will enable us in next lines to *override* the package source code.
- `src = pkgs.fetchgit {rev="..."; url="..."; sha256="...";};` defines a new source for the package we are overriding. Here, the source code will be fetched from a Git repository on url `https://framagit.org/mpoquet/simgrid.git` and commit `44bca631e482bd6e48a926393437d0959b661218`. The `sha256` attribute is a checksum used by Nix to make sure the fetched data is the expected one.

Note: Filling the `sha256` field can seem tricky for Nix newcomers. A fast way to do it is to:

1. Fill the field with a random SHA-256 string. For example, calling `echo simgrid | sha256sum` will generate a valid SHA-256 string.
2. Try to build your package (here, `nix-build -A my-custom-simgrid`). Nix will cry about hash mismatch.

   ```
   hash mismatch in fixed-output derivation '/nix/
   ↪storesk57v94s4y55b6r0xfzzy6g3sfsg20mi-simgrid-44bca63':
     wanted: sha256:1g3jwfnij8016b866frc8j146fp39ivlznm1ib726xjz700lr3kr
     got:    sha256:1fs0sdwx77yrakbffh00g7hxqladbjpqcs15lcx37viahjlk7fp0
   ```

3. Copy the `got` value computed by Nix into your Nix expression.

You can now modify the `batsim` package to use `my-custom-simgrid` instead of `simgrid`. This is done by changing the `simgrid` input when we *override* the Batsim package defined in NUR-Kapack, as shown in this diff:

```
-    batsim = (kapack.batsim.override { inherit debug simgrid; }).overrideAttrs␣
↪(attr: rec {
+    batsim = (kapack.batsim.override { inherit debug; simgrid=my-custom-simgrid; }).
↪overrideAttrs (attr: rec {
```

And that's it, the `batsim` package now uses the SimGrid version we just defined :).

If you want to use a local SimGrid version rather than one from a Git repository, the steps to follow are the same, you just need to change the source of your `my-custom-simgrid` package:

```
my-custom-simgrid = (kapack.simgrid-light.override { inherit debug; }).overrideAttrs␣
↪(attr: rec {
  src = /path/to/your/local/simgrid/source/repository;
});
```

23.8 Hacking Batsim and a scheduler at the same time

Similarly to *Changing Batsim dependencies*, it is quite common to work on both Batsim and a scheduler at the same time. This is notably the case when modifying the *Protocol*. This section shows how to hack batsched/pybatsim and Batsim at the same time.

A first simple way to do this is to bypass our Nix environments and put your own version of batsched/pybatsim in the `PATH` of the shell that launches the integration tests. The procedure to achieve this is very similar to what we have done in *Run tests from iteratively-built Batsim* so you can refer to it for details on how to setup your `PATH` environment variable. Here are short instructions on how to build your own local version of batsched and pybatsim:

- **Batsched** can be obtained from batsched's git repository and is built in the same way as Batsim. From the root of batsched's git repository, you can enter a shell able to build the batsched executable by calling `nix-shell -A batsched`. `meson build` should then generate a build directory. And finally, `ninja -C build` should compile a `batsched` executable in the build directory.
- **Pybatsim** can be obtained from pybatsim's git repository. As I write these lines, pybatsim does not have a clean Nix environment support for now, but you can enter a virtualenv to work on pybatsim with the following commands (from the root of pybatsim's git repository). First, either install python/virtualenv in your local machine or enter a Nix shell that has python and virtualenv: `nix-shell -p python3Packages.virtualenv`. Then, create a new virtualenv by calling `virtualenv venv` and set your environment variables by calling `source ./venv/bin/activate`. You can then build/install a local pybatsim by calling `pip install .`.

From there, hacking the environment of a shell that runs Batsim integration tests should be very easy for batsched, but it can be a bit tricky for pybatsim as it uses Python. Instead, you can decide to go for a Nix setup to change the versions of batsched or pybatsim, in a very similar fashion to what we did in *Changing Batsim dependencies*. Here is an example on what to add in the `jobs` Nix set in Batsim's `default.nix` to create your custom versions of batsched and pybatsim:

```
my-custom-batsched = (kapack.batsched.override { inherit debug; }).overrideAttrs␣
↪(attr: rec {
  src = /path/to/your/local/batsched/source/repository;
  preConfigure = "rm -rf build";
});

my-custom-pybatsim = kapack.pybatsim.overridePythonAttrs(attr: rec {
  src = /path/to/your/local/pybatsim/source/repository;
});
```

This Nix expression is very similar to what we have done and explained in *Changing Batsim dependencies*, so please refer to it for a detailed explanation of this snippet. A new Nix trick here is `overridePythonAttrs`, which does exactly the same as `overrideAttrs` but with the additional dark magic to make it work with Python. We also added `preConfigure = "rm -rf build";` in the `my-custom-batsched` package definition, this line makes sure the Nix build starts from a clean `build` directory (this does **not** removes the `build` directory in your local repository that contain batsched's sources, the only directory deleted by this command is in the temporary copy done by Nix when it builds the package). You can also of course use a git repository as the package source instead of a directory in your local filesystem (once again, see *Changing Batsim dependencies* for details).

Then, you can change the `integration_tests` attribute in `default.nix` to use your versions of batsched and pybatsim, as shown in the following diff:

```
    buildInputs = with pkgs.python37Packages; [
-     batsim batsched batexpe pkgs.redis
-     pybatsim pytest pytest_html pandas] ++
+     batsim my-custom-batsched batexpe pkgs.redis
```

(continues on next page)

(continued from previous page)

```
+     my-custom-pybatsim pytest pytest_html pandas] ++
    pkgs.lib.optional doValgrindAnalysis [ pkgs.valgrind ];
```

Now, whenever you run the `integration_tests` (`nix-build -A integration_tests`) or enter a shell to do so (`nix-shell -A integration_tests`), the environment should use your custom versions of batsched and pybatsim.

CHAPTER 24

How to run Batsim tests?

Note: Feel free to give a look at our *Continuous Integration* script to see how our robots run the tests.

24.1 Unit tests

Batsim unit tests are integrated into the Meson build system. Unit tests are also integrated in `default.nix`. Batsim can be built with unit tests: `nix-build -A batsim --arg doUnitTests true` (assuming that your current working directory is the root of Batsim's git repository).

Unit tests can also be run manually. In this case you can enter a dedicated environment to build batsim thanks to `nix-shell -A batsim` — or define your own environment. This enables incremental builds and can be convenient while your are editing Batsim's source code.

1. Tell Meson to compile unit tests by setting the `-Ddo_unit_tests` option when *configuring* your Meson build: `meson build -Ddo_unit_tests=true`.
2. Compile as usual (with Ninja): `ninja -C build`. This should generate an executable file `batunittest` in charge of running unit tests.
3. Run `batunittest` manually (`./build/batunittest`) or via Meson (`meson test -C build`).

24.2 Integration tests

Most Batsim tests are integration tests that do the following.

1. Execute a simulation instance on well-defined input and with a well-defined scheduler (including scheduler parameters).
2. Make sure that the execution went well (no infinite loop, no Batsim or scheduler crash…).
3. (Read simulation output files and check that they match some properties.)

Integration tests are written in Python with pytest in the `test` directory (from the root directory of Batsim's git repository). As in *Running your first simulation*, most of these tests use batsched and robin.

The simplest way to run the integration tests on the current local sources of Batsim is to run `nix-build -A integration_tests` from the root directory of Batsim's git repository. This will do the following.

1. (Recompile Batsim from local files if needed.)
2. Get into an environment with the local Batsim and the various test dependencies.
3. Run all integration tests.
4. Generate an HTML test report.
5. Put results in the *nix store* — accessible from a `result` symbolic link.

Results and logs should be convenient to explore from the HTML report. In other words: `firefox ./result/pytest_report.html`.

Alternatively, you can enter a shell where all the dependencies are available (`nix-shell -A integration_tests`) and call pytest manually from there. Please note that with this approach, you must reenter the shell whenever you modify Batsim's source code (as it needs to be compiled again). When you are working on a specific test, it can be useful to only run this test while compiling Batsim if needed before running it. More advanced `nix-shell` commands such as `nix-shell -A integration_tests --command "pytest test/test_nosched.py"` can be very useful in this case.

24.3 Other tests

The generation of Batsim's Doxygen documentation should not issue any warning. This can be checked manually or just by running `nix-build -A doxydoc` from Batsim's repository root directory.

CHAPTER 25

Continuous Integration

Batsim is tested under Gitlab's continuous integration system.

- Main CI script is `.gitlab-ci.yml` (from Batsim's repository root directory).
- CI logs are available on Batsim's Framagit Pipelines.
- Docker image used for tests is defined in `env/docker/Dockerfile` (from Batsim's repository root directory).
- Give a look at CI's script to reproduce locally. Enable Batsim's Cachix cache to not recompile dependencies: `cachix add batsim`.

Additionnally, a Batsim container is deployed on Dockerhub for each commit on the master branch. This is done on GitHub Actions whose script is defined in `build-docker-containers.yml` (from Batsim's repository root directory).

CHAPTER 26

Release procedure

Note: The following notations are used on this page.

- X.Y.Z stands for the new release.
- A.B.C stands for the previous release.

26.1 Any release

1. Determine what version number should be released according to Semantic Versioning. Major version should be changed whenever an incompatible change occurs on the CLI or the protocol.
2. Update the *Changelog*. All modifications since last release that may impact users should be given. `git diff vA.B.C [--stat]` can be helpful here.
3. Bump version wherever needed. `git grep 'A.B.C'` can be helpful here.
4. Tag new version with annotations: `git tag -a vX.Y.Z`. Put the changelog of the new release in the long description.
5. Push the tag on the *Continuous Integration* 's repo and make sure it passes. Pushing a tag can be done via `git push REMOTE vX.Y.Z`. If the tag is not validated by *Continuous Integration*, delete the tag (`git tag --delete vX.Y.Z && git push --delete REMOTE vX.Y.Z`), fix the issue and start again.
6. Push the tag on all remotes.

26.2 Main release (not for pre-releases)

1. Do all steps of *Any release*.

2. Update the *releases* branch. **Do not resolve the merge as a fast-forward!** This should be done via `git checkout releases && git merge --no-ff vX.Y.Z`. The goal is that the *releases* branch only contains released versions (`git log --first-parent --oneline releases`).
3. Push the *releases* branch on all remotes (`git push REMOTE releases`).
4. Create a new Batsim package in kapack.
    - The package name should follow Nix standard nomenclature (`nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -i batsim-X.Y.Z`).
    - The default `batsim` should point to the new package (`nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -iA batsim`).
    - **The previous Batsim package should remain accessible** (`nix-env -f https://github.com/oar-team/nur-kapack/archive/master.tar.gz -i batsim-A.B.C`).

CHAPTER 27

TODOs

27.1 Features TODOs

These are listed in our internal issues.

27.2 Documentation TODOs

Todo: Add more detailed to the example:

- the platform file
- the full workload file
- the launch command

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/batsim/checkouts/latest/docs/IO.rst, line 71.)

Todo:

- Finish to describe the Batsim energy model.
- Add instantiation examples.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/batsim/checkouts/latest/docs/input-platform.rst, line 68.)

Todo: We may think of more interesting things to plot while remaining simple. This is not easy on this toy workload though…

Maybe include the plot in this document if it is interesting.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/batsim/checkouts/latest/docs/tuto-result-analysis/tuto.rst, line 32.)

Todo: Introduce ViTE here and show an output example.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/batsim/checkouts/latest/docs/tuto-result-analysis/tuto.rst, line 55.)

Todo: Talk about Evalys / custom scripts

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/batsim/checkouts/latest/docs/tuto-result-analysis/tuto.rst, line 63.)

Todo: Hacking guidelines are not written yet about these projects.

Do not hesitate to *Contact us* about it.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/batsim/checkouts/latest/docs/tuto-sched-implem/tuto.rst, line 44.)