WRENCH

Release 2.2-dev

WRENCH Team

Jul 20, 2023

QUICKSTART

WRENCH

WRENCH is an open-source framework designed to make it easy for users to develop accurate and scalable simulators of distributed computing applications, systems, and platforms. It has been used for research, development, and education. WRENCH capitalizes on recent and critical advances in the state of the art of simulation of distributed computing scenarios. Specifically, WRENCH builds on top of the open-source SimGrid simulation framework. SimGrid enables the simulation of distributed computing scenarios in a way that is accurate (via validated simulation models), scalable (low ratio of simulation time to simulated time, ability to run large simulations on a single computer with low compute, memory, and energy footprints), and expressive (ability to simulate arbitrary platform, application, and execution scenarios). WRENCH provides directly usable high-level simulation abstractions, which all use SimGrid as a foundation, to make it possible to implement simulators of complex scenarios with minimal development effort.

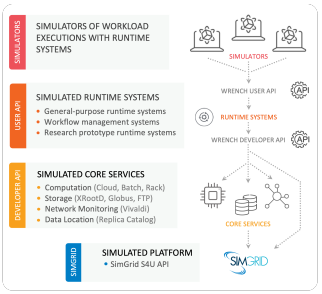
In a nutshell, WRENCH makes it possible to:

- Develop in-simulation implementations of runtime systems that execute application workloads on distributed hardware platforms managed by various software services commonly known as Cyberinfrastructure (CI) services.
- Quickly, scalably, and accurately simulate arbitrary application and platform scenarios for these runtime system implementation.

WRENCH is an *open-source C++ library* for developing simulators. It is neither a graphical interface nor a standalone simulator. WRENCH exposes several high-level simulation abstractions to provide high-level **building blocks** for developing custom simulators.

WRENCH comprises four distinct layers:

- **Top-Level Simulation:** A top-level set of abstractions to instantiate a simulator that simulates the execution of a runtime system that executes some application workload on some distributed hardware platform whose resources are accessible via various services.
- **Simulated Execution Controller:** An in-simulation implementation of a runtime system designed to execute some application workload.
- **Simulated Core Services:** Abstractions for simulated cyberinfrastructure (CI) components that can be used by the runtime system to execute application workloads (compute services, storage services, network proximity services, data location services, etc.).
- **Simulation Core:** All necessary simulation models and base abstractions (computing, communicating, storing), provided by SimGrid.



CHAPTER

ONE

THREE CLASSES OF USERS

On can distinguish three kinds of WRENCH users:

- **Runtime System Users** use WRENCH to simulate application workload executions using an already available, in-simulation implementation of a runtime system that uses Core Services to execution that workload.
- **Runtime System Developers/Researchers** use WRENCH to prototype and evaluate runtime system designs and/or to investigate and evaluate novel algorithms to be implemented in a runtime system.
- **Internal Developers** contribute to the WRENCH code, typically by implementing new Core Services.

1.1 Three Levels of API Documentation

The WRENCH library provides three *incremental* levels of documentation, each targeting an API level:

User: This level targets users who want to use WRENCH for simulating the execution of application workloads using already implemented runtime systems. *Users* are NOT expected to develop new simulation abstractions or algorithms. Instead, they only use available simulation components as high-level building blocks to quickly build simulators. These simulators can involve as few as a 50-line of C++ code.

Developer: This level targets *runtime system developers and researchers* who work on developing novel runtime system designs and algorithms. In addition to documentation for all simulation components provided at the *User* level, the *Developer* documentation includes detailed documentation for interacting with simulated Core Services. There are **two Developer APIs**. The most generic API is called the *Action API*, and allows developers to describe and execution application workloads that consist of arbitrary "actions". The *Workflow API* is specifically designed for those developers that implement workflow runtime systems (also known as Workflow Management Systems, or WMSs), and as such is provides a Workflow abstraction that these developers will find convenient. All details are provided in the rest of the documentation.

Internal: This level targets those users who want to contribute code to WRENCH. It provides, in addition to both levels above, detailed documentation for all WRENCH classes including binders to SimGrid. This is the API needed to, for instance, implement new Core Services.

CHAPTER

TWO

GET IN TOUCH

The main channel to reach the WRENCH team is via the support email: support@wrench-project.org.

Bug Report / Feature Request: our preferred channel to report a bug or request a feature is via WRENCH's Github Issues Track.

2.1 Installing WRENCH

2.1.1 Prerequisites

WRENCH is developed in C++. The code follows the C++14 standard, and thus older compilers may fail to compile it. Therefore, we strongly recommend users to satisfy the following requirements:

- **CMake** - version 3.10 or higher

And, one of the following: - **g++** - version 7.5 or higher - **clang** - version 9.0 or higher

Required Dependencies

- SimGrid – version 3.34
- JSON for Modern C++ – version 3.9.0 or higher

(See the *Installation Troubleshooting* section below if encountering difficulties installing dependencies)

Optional Dependencies

- Google Test – version 1.8 or higher (only required for running tests)
- Doxygen – version 1.8 or higher (only required for generating documentation)
- Sphinx - version 4.5 or higher along with the following Python packages: `pip3 install sphinx-rtd-theme breathe recommonmark` (only required for generating documentation)
- Batsched – version 1.4 - useful for expanded batch-scheduled resource simulation capabilities
- Asio - tag 1.28.0 or later (only required for building *wrench-daemon*, WRENCH's REST API daemon)

2.1.2 Source Install

Building WRENCH

You can download the `wrench-<version>.tar.gz` archive from the GitHub releases page. Once you have installed dependencies (see above), you can install WRENCH as follows:

```
tar xf wrench-<version>.tar.gz
cd wrench-<version>
mkdir build
cd build
cmake ..
make -j8
make install # try "sudo make install" if you do not have write privileges
```

If you want to see actual compiler and linker invocations, add `VERBOSE=1` to the compilation command:

```
make -j8 VERBOSE=1
```

To enable the use of Batsched (provided you have installed that package, see above):

```
cmake -DENABLE_BATSCHED=on .
```

If you want to stay on the bleeding edge, you should get the latest git version, and recompile it as you would do for an official archive:

```
git clone https://github.com/wrench-project/wrench
```

Compiling and running unit tests

Building and running the unit tests, which requires Google Test, is done as:

```
make -j8 unit_tests
./unit_tests
```

Compiling and running examples

Building the examples is done as:

```
make -j8 examples
```

All binaries for the examples are then created in subdirectories of `build/examples/`

Installation Troubleshooting

Could NOT find PkgConfig (missing: PKG_CONFIG_EXECUTABLE)

- This error on MacOS is because the `pkg-config` package is not installed
- Solution: install this package
  - MacPorts: `sudo port install pkg-config`
  - Brew: `sudo brew install pkg-config`

Could not find libgfortran when building the SimGrid dependency

- This is an error that sometimes occurs on MacOS
- A quick fix is to disable the SMPI feature of SimGrid when configuring it: `cmake -Denable_smpi=off .`

2.1.3 Docker Containers

WRENCH is also distributed in Docker containers. Please, visit the WRENCH Repository on Docker Hub to pull WRENCH's Docker images.

The `latest` tag provides a container with the latest WRENCH release:

```
docker pull wrenchproject/wrench
# or
docker run --rm -it wrenchproject/wrench /bin/bash
```

The `unstable` tag provides a container with the (almost) current code in the GitHub's `master` branch:

```
docker pull wrenchproject/wrench:unstable
# or
docker run --rm -it wrenchproject/wrench:unstable /bin/bash
```

Additional tags are available for all WRENCH releases.

2.2 Getting started

Once you have installed the WRENCH library, following the instructions on the *installation page*, you are ready to create a WRENCH simulator. **Information on what can be simulated and how to do it are provided in the *WRENCH 101* and *WRENCH 102* pages. This page is only about the logistics of setting up a simulator project.**

2.2.1 Using the WRENCH initialization tool

The `wrench-init` tool is a project generator built with WRENCH, which creates a simple project structure as follows:

```
project-folder/
├── CMakeLists.txt
├── CMakeModules
│   └── FindSimGrid.cmake
│   └── FindWRENCH.cmake
├── src/
│   ├── Simulator.cpp
│   ├── Controller.cpp
├── include/
│   └── Controller.h
├── build/
└── data/
    └── platform.xml
```

The `Simulator.cpp` source file contains the `main()` function of the simulator, which initializes a simulated platform and services running on this platform; `Controller.h` and `Controller.cpp` contain the implementation of an execution controller, which executes a workflow on the available services. The simulator takes as command-line argument a path to a platform description file in XML, which is available in `data/platform.xml`. These files provide the minimum necessary implementation for a WRENCH-enabled simulator.

The `wrench-init` tool only requires a single argument, the name of the folder where the project skeleton will be generated:

```
$ wrench-init <project_folder>
```

Additional options supported by the tool can be found by using the `wrench-init --help` command.

Of course, you do not have to use `wrench-init`, especially if you are used to creating your own CMake projects. But you still may want to look at the `CMakeLists.txt` file generated by `wrench-init`. In particular, note that `CMakeLists.txt` uses the `FindSimgrid.cmake` and `FindWRENCH.cmake` files, which are placed by `wrench-init` in the `CMakeModules` directory.

2.2.2 Example WRENCH simulators

The examples in the `examples` directory provide good starting points for developing your own simulators. Examples are provided for the generic "action" API as well as for the "workflow" API, and are built along with the WRENCH library and tools. See the `examples/README.md` file for a brief description of all examples. Examples can be built by typing `make examples` in the build directory.

For instance, the `examples/action_api/bare-metal-bag-of-actions` example can be executed as:

```
$ wrench-example-bare-metal-bag-of-actions 6 two_hosts.xml --log=custom_wms.
↪threshold=info
```

(File `two_hosts.xml` is in the `examples/action_api/bare-metal-bag-of-actions` directory.) You should see some output in the terminal. The output in white is produced by the simulator's main function. The output in green is produced by the execution controller implemented with the WRENCH developer API.

Although you can inspect the codes of the examples on your own, we highly recommend that you go through the *Simulation 101*, *WRENCH 101*, and *WRENCH 102* pages first. These pages make direct references to the examples, a description of which is available in `examples/README.md` in the WRENCH distribution.

2.3 Simulation 101

This page provides a gentle introduction to the simulation of parallel and distributed executions, as enabled by WRENCH. This content is intended for users who have never implemented (or even thought of implementing) a simulator.

2.3.1 Simulation Overview

A simulator is a software artifact that mimics the behavior of some system of interest. In the context of the WRENCH project, the systems of interest are parallel and distributed platforms on which various software runtime systems are deployed by which some application workload is to be executed. For instance, the platform could be a homogeneous cluster with some network attached storage, the software runtime systems could be a batch scheduler and a file server that controls access to the network attached storage, and the application workload could be a scientific workflow. The system could be much more complex, with different kinds of runtime systems running on hardware or virtualized resources connected over a wide-area network.

2.3.2 Simulated Platform

A simulated platform consists of a set of computers, or, **hosts**. These hosts can have various characteristics (e.g., number of cores, clock rate). Each host can have one or more **disks** attached to it, on which data can be stored and accessed. The hosts are interconnected with each other over a network (otherwise this would not be parallel and distributed computing). The network is a set of network **links**, each with some latency and bandwidth specification. Two hosts are connected via a network path, which is simply a sequence of links through which messages between the two hosts are routed.

The above concepts allow us to describe a simulated platform that can resemble real-world, either current or upcoming, platforms. Many more details and features of the platform can be described, but the above concepts gives us enough of a basis for everything that follows. Platform description in WRENCH is based on the platform description capabilities in SimGrid: a platform can be described in an XML file or programmatically (see more details on the *WRENCH 101* page).

2.3.3 Simulated Processes

The execution of processes (i.e., running programs) can be simulated on the hosts of the simulated platform. These processes can execute arbitrary (C++) code and also place calls to WRENCH to simulate usage of the platform resources (i.e., now I am computing, now I am sending data to the network, now I am reading data from disk, now I am creating a new process, etc.). As a result, the speed of the execution of these processes is limited by the characteristics of the hardware resources in the platform, and their usage by other processes. Process executions proceed through simulated time until the end of the simulation, e.g., when the application workload of interest has completed. At that point, the simulator can, for instance, print the simulated time.

At this point, you may be thinking: "Are you telling me that I need to implement a bunch of simulated processes that do things and talk to each other? My system is complicated and do not even know all the processes I would need to simulate! There is no way I can do this!". **And you would be right.** It is true that any parallel and distributed system of interest is, at its most basic level, just a set of processes that compute, read/write data, and send/receive messages. But it is a lot of work to implement a simulator of a complex system at such a low level. This is where WRENCH comes in.

2.3.4 Simulated Services

WRENCH comes with a large set of already-implemented **services**. A service is a set of one or more running simulated processes that simulate a software runtime system that is commonly useful and used for parallel and distributed computing. The two main kinds of services are *compute services* and *storage services*, but there are others (all detailed on the *WRENCH 101* page).

A compute service is a runtime system to which you can say "run this computation" and it replies either "ok, I will run it" or "I cannot". If it can run it, then later on it will tell you either "It is done" or "It is failed". And that is it. Underneath, this entails all kinds of processes that compute, communicate with each other, and start other processes. This complexity is all abstracted away by the service, which exposes a simple, high-level, easy-to-understand API. For instance, in our example earlier we mentioned a batch scheduler. For HPC (High Performance Computing), this is popular runtime system that manages the execution of jobs on a set of compute nodes on some fast local network, i.e., a cluster. In the real-world, a batch scheduler consists of many running processes (a.k.a. daemons) running on the cluster, implements sophisticated algorithms to decide which job should run next, makes sure jobs do not run on the same cores, etc. WRENCH provides an already-implemented compute service called a `wrench::BatchComputeService` that does all this for you, under the cover.

For example, the well-known batch scheduler Slurm uses several daemons to schedule and manage jobs(e.g., the process **slurmd** runs on each compute node and one **slurmctld** daemon controls everything). In this example, an instance of `wrench::BatchComputeService` could represent one Slurm cluster with one **slurmctld** process and multiple **slurmd** processes.

A storage service is a runtime system to which you can say "here is some data I want you to store", "I want to read some bytes from that data I stored before", "Do you have this data?", etc. A storage service in the real world consists of several processes (e.g., to handle bounded numbers of concurrent reads/writes from different clients) and can use non-trivial algorithms (e.g., for overlapping network communication and disk accesses). Here again, WRENCH comes with an already-implemented storage service called `wrench::SimpleStorageService` that does all this for you and comes with a straightforward, high-level API. Note that a storage service does not provide by default capabilities traditionally offered by parallel file systems such as Lustre (i.e., no stripping among storage nodes, no dedicated metadata servers). If you want to model such storage back-end, you can do it by extending the `wrench::SimpleStorageService`.

Each service in WRENCH comes with configurable *properties*, that are well-documented and can be used to specify particular features and/or behaviors (e.g., a specific scheduling algorithm for a given `wrench::BatchComputeService`). Each service also comes with *configurable message payloads*, which specify the size in bytes of the control messages that underlying processes exchange with each other to implement the service's functionality. In the real-world, the processes that comprise a service exchange various messages, and in WRENCH you get to specify the size of all these messages (the larger the sizes the longer the simulated communication times). See more about *Service Customization* on the *WRENCH 101* page.

When the simulator is done, the **calibration** phase begins. The **calibration** step is crucial to ensure that your simulator accurately approximate the performance of the application you study on the target platform. Basically, calibrating a simulator implies that you fine-tune the simulator to approximate the real performance of the target application when running on the modeled platform. *Payloads* and *properties* play a central role in this calibration step as they control the weight of many important actions (for example, how much overhead when reading a file from a storage service?).

2.3.5 Simulated Controller

As you recall, the goal of a WRENCH simulator is to simulate the execution of some application workload. And so far, we have not said much about this workload or about how one goes about simulating its execution. So let's…

An application workload is executed using the services deployed on the platform. To do so, you need to implement one process called an **execution controller**. This process invokes the services to execute the application workload, whatever that workload is. Say, for instance, that your application workload consists in performing some amount of computation based on data in some input file. The controller should ask a compute service to start a job to perform the computation, while reading the input from some storage service that stores the input file. Whenever the compute service replies that the computation has finished, then the execution controller's work is done.

The execution controller is the core of the simulator, as it is where you implement whatever algorithm/strategy you wish to simulate for executing the application workload. At this point the execution controller likely seems a bit abstract. But we would not say more about it until you get to the *WRENCH 102* page, which is exclusively about the controller.

2.3.6 What's next

At this point, you should be able to jump into *WRENCH 101*!

2.4 WRENCH 101

This page provides high-level and detailed information about what WRENCH simulators can simulate and how they do it. Full API details are provided in the *User API Reference*. See the relevant pages for instructions on how to *install WRENCH* and how to *setup a simulator project*.

2.4.1 10,000-ft view of a WRENCH simulator

A WRENCH simulator can be as simple as a single `main()` function that creates a platform to be simulated (the hardware) and a set of services that run on the platform (the software). These services correspond to software that knows how to store data, perform computation, and many other useful things that real-world cyberinfrastructure services can do.

The simulator then creates a special (simulated) process called an *execution controller*. An execution controller interacts with the services running on the platform to execute some application workload of interest, whatever that workflow is. The execution controller is implemented using the *WRENCH Developer API*, as discussed in the *WRENCH 102* page.

The simulation is then launched via a single call (`wrench::Simulation::launch()`), and returns only once the execution controller has terminated (after completing or failing to complete whatever it wanted to accomplish).

2.4.2 1,000-ft view of a WRENCH simulator

In this section, we dive deeper into what it takes to implement a WRENCH simulator. *To provide context, we refer to the example simulator in the* `examples/action_api/multi-action-multi-job` *directory of the WRENCH distribution*. This simulator simulates the execution of a few jobs, each of which consists of one or more actions, on a 4-host platform that runs a couple of compute services and storage services. Although other examples are available (see `examples/README.md`), this simple example is sufficient to showcase most of what a WRENCH simulator does, which consists in going through the steps below. Note that all simulator codes in the `examples` directory contain extensive comments.

Step 0: Include wrench.h

For ease of use, all WRENCH abstractions in the *WRENCH User API* are available through a single header file:

```
#include <wrench.h>
```

Step 1: Create and initialize a simulation

The state of a WRENCH simulation is defined by the `wrench::Simulation` class. A simulator must create an instance of this class by calling `wrench::Simulation::createSimulation()` and initialize it with the `wrench::Simulation::init()` member function. The `multi-action-multi-job` simulator does this as follows:

```
auto simulation = wrench::Simulation::createSimulation();
    simulation->init(&argc, argv);
```

Note that this member function takes in the command-line arguments passed to the main function of the simulator. This is so that it can parse WRENCH-specific and SimGrid-specific command-line arguments. (Recall that WRENCH is based on SimGrid.) Two useful such arguments are `--wrench-help`, which displays a WRENCH help message, and `--help-simgrid`, which displays an extensive SimGrid help message. Another one is `--wrench-full-log`, which displays full simulation logs (see below for more details).

Step 2: Instantiate a simulated platform

This is done with the `wrench::Simulation::instantiatePlatform()` method. There are two versions of this method. The **first version** takes as argument a SimGrid virtual platform description file, we defines all the simulated hardware (compute hosts, clusters of hosts, storage resources, network links, routers, routes between hosts, etc.). The bare-metal-chain simulator comes with a platform description file, `examples/action_api/multi-action-multi-job/four_hosts.xml`, which we include here:

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
    <zone id="AS0" routing="Full">

        <!-- The host on which the Controller will run -->
        <host id="UserHost" speed="10Gf" core="1">
        </host>

        <!-- The host on which the bare-metal compute service will run and also run jobs-
↪->
        <host id="ComputeHost1" speed="35Gf" core="10">
            <prop id="ram" value="16GB" />
        </host>

        <!-- Another host on which the bare-metal compute service will be able to run␣
↪jobs -->
        <host id="ComputeHost2" speed="35Gf" core="10">
            <prop id="ram" value="16GB" />
        </host>

        <!-- The host on which the first storage service will run -->
        <host id="StorageHost1" speed="10Gf" core="1">
```

(continues on next page)

(continued from previous page)

```
            <disk id="hard_drive" read_bw="100MBps" write_bw="100MBps">
                <prop id="size" value="5000GiB"/>
                <prop id="mount" value="/"/>
            </disk>
        </host>

        <!-- The host on which the second storage service will run -->
        <host id="StorageHost2" speed="10Gf" core="1">
            <disk id="hard_drive" read_bw="200MBps" write_bw="200MBps">
                <prop id="size" value="5000GiB"/>
                <prop id="mount" value="/"/>
            </disk>
        </host>

        <!-- The host on which the cloud compute service will run -->
        <host id="CloudHeadHost" speed="10Gf" core="1">
            <disk id="hard_drive" read_bw="100MBps" write_bw="100MBps">
                <prop id="size" value="5000GiB"/>
                <prop id="mount" value="/scratch/"/>
            </disk>
        </host>

        <!-- The host on which the cloud compute service will start VMs -->
        <host id="CloudHost" speed="25Gf" core="8">
            <prop id="ram" value="16GB" />
        </host>

        <!-- A network link shared by EVERY ONE-->
        <link id="network_link" bandwidth="50MBps" latency="1ms"/>

        <!-- The same network link connects all hosts together -->
        <route src="UserHost" dst="ComputeHost1"> <link_ctn id="network_link"/> </route>
        <route src="UserHost" dst="ComputeHost2"> <link_ctn id="network_link"/> </route>
        <route src="UserHost" dst="StorageHost1"> <link_ctn id="network_link"/> </route>
        <route src="UserHost" dst="StorageHost2"> <link_ctn id="network_link"/> </route>
        <route src="UserHost" dst="CloudHeadHost"> <link_ctn id="network_link"/> </route>
        <route src="ComputeHost1" dst="StorageHost1"> <link_ctn id="network_link"/> </
↪route>
        <route src="ComputeHost2" dst="StorageHost2"> <link_ctn id="network_link"/> </
↪route>
        <route src="CloudHeadHost" dst="CloudHost"> <link_ctn id="network_link"/> </
↪route>
        <route src="StorageHost1" dst="CloudHost"> <link_ctn id="network_link"/> </route>
        <route src="StorageHost2" dst="CloudHost"> <link_ctn id="network_link"/> </route>

    </zone>
</platform>
```

This file defines a platform with several hosts, each with some number of cores and a core speed. Some hosts have a disk attached to them, some declare a RAM capacity. The platform also declares a single network link with a particular latency and bandwidth, and routes between some of the hosts (over that one link). We refer the reader to platform description files in other examples in the `examples` directory and to the SimGrid documentation for more information on how to create platform description files. There are many possibilities for defining complex platforms at will. The

bare-metal-chain simulator takes the path to the platform description as its 1st (and only) command-line argument and thus instantiates the simulated platform as:

```
simulation.instantiatePlatform(argv[1]);
```

The **second version** of the `wrench::Simulation::instantiatePlatform()` method takes as input a function that creates the platform description programmatically using the SimGrid platform description API. The example in `examples/workflow_api/basic-examples/bare-metal-bag-of-tasks-programmatic-platform` shows how the XML platform description in `examples/workflow_api/basic-examples/bare-metal-bag-of-tasks/two_hosts.xml` can be implemented programmatically. (Note that this example passes a functor to `wrench::Simulation::instantiatePlatform()` rather than a plain lambda.)

Step 3: Instantiate services on the platform

While the previous step defines the hardware platform, this step defines what software services run on that hardware. The `wrench::Simulation::add()` member function is used to add services to the simulation. Each class of service is created with a particular constructor, which also specifies host(s) on which the service is to be started. Typical kinds of services include compute services, storage services, and file registry services (see *below* for more details).

The multi-action-multi-job simulator instantiates four services. The first one
is a storage service:

```
auto storage_service_1 = simulation->add(new wrench::SimpleStorageService("StorageHost1",
↪ {"/"}, {{wrench::SimpleStorageServiceProperty::BUFFER_SIZE, "50MB"}}, {}));
```

The `wrench::SimpleStorageService` class implements a simulation of a remotely-accessible storage service on which files can be stored, copied, deleted, read, and written. In this particular case, the storage service is started on host `StorageHost1`. It uses storage mounted at `/` on that host (which corresponds to the mount path of a disk, as seen in the XML platform description). The last two arguments, as for the compute services, are used to configure particular properties of the service. In this case, the service is configured to use a 50-MB buffer size to pipeline network and disk accesses (see details in *this section below*).

The second service is a another storage service that runs on host `StorageHost2`.

The third service is a compute service:

```
auto baremetal_service = simulation->add(new wrench::BareMetalComputeService(
↪"ComputeHost1", {{"ComputeHost1"}, {"ComputeHost2"}}, "", {}, {}));
```

The `wrench::BareMetalComputeService` class implements a simulation of a compute service that greedily runs jobs submitted to it. You can think of it as a compute server that simply fork-execs (possibly multi-threaded) processes upon request, only ensuring that physical RAM capacity is not exceeded. In this particular case, the compute service is started on host `ComputeHost1`. It has access to the compute resources of that same host as well as that of a second host `ComputeHost2` (2nd argument is a list of available compute hosts). The third argument corresponds to the path of some scratch storage, i.e., storage in which data can be stored temporarily while a job runs. In this case, the scratch storage specification is empty as host `ComputeHost1` has no disk attached to it. The last two arguments are `std::map` objects (in this case both empty), that are used to configure properties of the service (see details in *this section below*).

The fourth service is a cloud compute service:

```
auto cloud_service = simulation->add(new wrench::CloudComputeService("CloudHeadHost", {
↪"CloudHost"}, "/scratch/", {}, {}));
```

The `wrench::CloudComputeService` implements a simulation of a cloud platform on which virtual machine (VM) instances can be created, started, used, and shutdown. The service runs on host `CloudHeadHost` and has access to the compute resources on host `CloudHost`. Unlike the previous service, this service has scratch space, at path `/scratch`

on the disk attached to host `CloudHost` (as seen in the XML platform description). Here again, the last two arguments are used to configure properties of the service.

Step 4: Instantiate at least one Execution controller

At least one *execution controller* must be created and added to the simulation. This is a special service that is in charge of executing an application workload on the platform. It is implemented as a class that derives from `wrench::ExecutionController` and override its constructor as well as its `main()` method. This method is implementing using the *WRENCH Developer API*.

The example in `examples/action_api/multi-action-multi-job` does this as follows:

```
auto wms = simulation->add(new wrench::MultiActionMultiJobController(baremetal_service,␣
↪cloud_service, storage_service_1, storage_service_2, "UserHost"));
```

This creates an execution controller and passes to its constructor the services created before, and the host on which it is supposed to execute. Class `wrench::MultiActionMultiJobController` is of course provided with the example. See the *WRENCH 102* page for information on how to implement an execution controller.

One important question is how to specify an *application workload* and tell the execution controller to execute it. This is completely up to the developer, and in this example the execution controller creates a different number of tasks to creates files, file read actions, file write actions, and compute actions to be executed as part of various jobs (see the implementation of `wrench::MultiActionMultiJobController`). All the examples in the `examples/action_api` directory do this in different ways. *However*, many users are interested in **workflow applications**, for this reason, WRENCH provides a `wrench::Workflow` class that has member functions to manually create tasks and files and add them to a workflow. The use of this class is shown in all the examples in directory `examples/workflow_api`. The `wrench::Workflow` class also provides member functions to import workflows from workflow description files in standard JSON format. Note that an execution controller that executes a workflow is often called a Workflow Management System (WMS). This is why many execution controllers in the examples in directory `examples/workflow_api` have WMS in their class names.

Step 5: Launch the simulation

This is the easiest step, and is done by simply calling `wrench::Simulation::launch()`:

```
simulation.launch();
```

This call checks the simulation setup and blocks until the execution controller terminates.

Step 6: Process simulation output

The processing of simulation output is up to the user as different users are interested in different output. For instance, the examples in directory `examples/action_api` merely print some information to the terminal. But this information could be collected in data structures, output to files, etc. This said, WRENCH provides a `wrench::Simulation::getOutput()` member function that returns an instance of class `wrench::SimulationOutput`. Note that there are member functions to configure the type and amount of output generated (see the `wrench::SimulationOutput::enable*Timestamps()` member functions). `wrench::SimulationOutput` has a templated `wrench::SimulationOutput::getTrace()` member function to retrieve traces for various information types. This is exemplified in several of the example simulators in the `examples/workflow_api` directory. Note that many of the timestamp types have to do with the execution of workflow tasks, as defined using the `wrench::Workflow` class.

Another kind of output is (simulated) energy consumption. WRENCH leverages SimGrid's energy plugin, which provides accounting for computing time and dissipated energy in the simulated platform. SimGrid's energy plugin requires host `pstate` definitions (levels of performance, CPU frequency) in the XML platform description file. The `wrench::Simulation::getEnergyConsumed()` member function returns energy consumed by all hosts in the platform. **Important:** The energy plugin is NOT enabled by default in WRENCH simulations. To enable it, pass the `--wrench-energy-simulation` command line option to the simulator. See `examples/workflow_api/basic-examples/cloud-bag-of-tasks-energy` for an example simulator that makes use of this plugin (and an example platform description file that defines host power consumption profiles).

It is also possible to dump all simulation output to a JSON file. This is done with the `wrench::SimulationOutput::dump*JSON()` member functions. The documentation of each member function details the structure of the JSON output, in case you want to parse/process the JSON by hand. See the API documentation of the `wrench::SimulationOutput` class for all details.

Alternatively, you can run the installed `wrench-dashboard` tool, which provides interactive visualization/inspection of the generated JSON simulation output. You can run the dashboard for the JSON output generated by the example simulators in `examples/workflow_api/basic-examples/bare-metal-bag-of-task` and `examples/workflow_api/basic-examples/cloud-bag-of-task`. These simulators produce a JSON file in `/tmp/wrench.json`. Simply run the command `wrench-dashboard`, which pops up a Web browser window in which you simply upload the `/tmp/wrench.json` file.

We find that most users end up doing their own, custom simulation output generation since they are the ones who know what they are interested in.

2.4.3 Available services

Below is the list of services available to-date in WRENCH. Click on the corresponding links for more information on what these services are and on how to create them.

- **Compute Services**: These are services that know how to compute workflow tasks:
  - Bare-metal Servers
  - Cloud Platforms
  - Virtualized Cluster Platforms
  - Batch-scheduled Clusters
  - HTCondor
- **Storage Services**: These are services that know how to store and give access to workflow files:
  - Simple Storage Service
  - XRootD Storage Service
- **File Registry Services**: These services, also known as *replica catalogs*, are simply databases of <`filename, list of locations`> key-values pairs of the storage services on which copies of files are available.
  - File Registry Service
- **Network Proximity Services**: These are services that monitor the network and maintain a database of host-to-host network distances:
  - Network Proximity Service
- **EnergyMeter Services**: These services are used to periodically measure host energy consumption and include these measurements in the simulation output:
  - Energy Meter Service

- **BandwidthMeter Services**: These services are used to periodically measure network links' bandwidth usage and include these measurements in the simulation output:
  - Bandwidth Meter Service

2.4.4 Customizing services

Each service is customizable by passing to its constructor a *property list*, i.e., a key-value map where each key is a property and each value is a string. Each service defines a property class. For instance, the `wrench::Service` class has an associated `wrench::ServiceProperty` class, the `wrench::ComputeService` class has an associated `wrench::ComputeServiceProperty` class, and so on at all levels of the service class hierarchy.

The API documentation for these property classes explains what each property means, what possible values are, and what default values are. Other properties have more to do with what the service can or should do when in operation. For instance, the `wrench::BatchComputeServiceProperty` class defines a `wrench::BatchComputeServiceProperty::BATCH_SCHEDULING_ALGORITHM` which specifies what scheduling algorithm a batch service should use for prioritizing jobs. All property classes inherit from the `wrench::ServiceProperty` class, and one can explore that hierarchy to discover all possible (and there are many) service customization opportunities.

Finally, each service exchanges messages on the network with other services (e.g., an execution controller sends a "do some work for me" messages to compute services). The size in bytes, or payload, of all messages can be customized similarly to the properties, i.e., by passing a key-value map to the service's constructor. For instance, the `wrench::ServiceMessagePayload` class defines a `wrench::ServiceMessagePayload::STOP_DAEMON_MESSAGE_PAYLOAD` property which can be used to customize the size, in bytes, of the control message sent to the service daemon (that is the entry point to the service) to tell it to terminate. Each service class has a corresponding message payload class, and the API documentation for these message payload classes details all messages whose payload can be customized.

2.4.5 Customizing logging

When running a WRENCH simulator you may notice that there is no logging output. By default logging output is disabled, but it is often useful to enable it (remembering that it can slow down the simulation). WRENCH's logging system is a thin layer on top of SimGrid's logging system, and as such is controlled via command-line arguments.

The `bare-metal-chain` example simulator can be executed as follows in the `examples/action_api/bare-metal-bag-of-actions` subdirectory of the build directory (after typing `make examples` in the build directory):

```
./wrench-example-bare-metal-bag-of-tasks 10 ./four_hosts.xml
```

The above generates almost no output to the terminal whatsoever. It is possible to enable some logging to the terminal. It turns out the execution controller class in that example (`TwoTasksAtATimeExecutionController.cpp`) defines a logging category named `custom_controller` (see one of the first lines of `examples/action_api/bare-metal-bag-of-actions/TwoActionsAtATimeExecutionController.cpp`), which can be enabled as:

```
./wrench-example-bare-metal-bag-of-tasks 10 ./four_hosts.xml --log=custom_controller.
↪threshold=info
```

You will now see some (green) logging output that is generated by the execution controller implementation. It is typical to want to see these messages as the controller is the brain of the application workload execution.

One can disable the coloring of the logging output with the `--wrench-no-color` argument:

```
./wrench-example-bare-metal-bag-of-tasks 10 ./four_hosts.xml --log=custom_controller.
↪threshold=info --wrench-no-color
```

Disabling color can be useful when redirecting the logging output to a file.

Enabling all logging is done with the argument `--wrench-full-log`:

```
./wrench-example-bare-metal-bag-of-tasks 10 ./four_hosts.xml --wrench-full-log
```

The logging output now contains output produced by all the simulated running processed. More details on logging capabilities are displayed when passing the `--help-logs` command-line argument to your simulator. Log category names are attached to `*.cpp` files in the simulator code, the WRENCH code, and the SimGrid code. Using the `--help-log-categories` command-line argument shows the entire log category hierarchy (which is huge).

See the Simgrid logging documentation for all details.

2.5 WRENCH 102

In WRENCH's terminology, and *execution controller* is software that makes all decisions and takes all actions for executing some application workflow using cyberinfrastructure services. It is thus a crucial component in every WRENCH simulator. WRENCH does not provide any execution controller implementation, but provides the means for developing custom ones. This page is meant to provide high-level and detailed information about implementing an execution controller in WRENCH. Full API details are provided in the *Developer API Reference*.

2.5.1 Basic blueprint for an execution controller implementation

An execution controller implementation needs to use many WRENCH classes, which are accessed by including a single header file:

```
#include <wrench-dev.h>
```

An execution controller implementation must derive the `wrench::ExecutionController` class, which means that it must override several the virtual `main()` member function. A typical such implementation of this function goes through a simple loop as follows:

```
// A) create/retrieve application workload to execute
// B) obtain information about running services
while (application workload execution has not completed/failed) {
  // C) interact with services
  // D) wait for an event and react to it
}
```

In the next three sections, we give details on how to implement the above. To provide context, we make frequent references to the execution controllers implemented as part of the example simulators in the `examples/` directory. Afterwards are a few sections that highlight features and functionality relevant to execution controller development.

2.5.2 A) Finding out information about running services

Services that the execution controller can use are typically passed to its constructor. Most service classes provide member functions to get information about the capabilities and properties of the services. For instance, a `wrench::ComputeService` has a `wrench::ComputeService::getNumHosts()` member function that returns how many compute hosts the service has access to in total. A `wrench::StorageService` has a `wrench::StorageService::getFreeSpace()` member function to find out how many bytes of free space are available on it. And so on...

To take a concrete example, consider the execution controller implementation in `examples/workflow_api/basic-examples/batch-bag-of-tasks/TwoTasksAtATimeBatchWMS.cpp`. This WMS finds out the compute speed of the cores of the compute nodes available to a `wrench::BatchComputeService` as:

```
double core_flop_rate = (*(batch_service->getCoreFlopRate().begin())).second;
```

Member function `wrench::ComputeService::getCoreFlopRate()` returns a map of core compute speeds indexed by hostname (the map thus has one element per compute node available to the service). Since the compute nodes of a batch compute service are homogeneous, the above code simply grabs the core speed value of the first element in the map.

It is important to note that these member functions actually involve communication with the service, and thus incur overhead that is part of the simulation (as if, in the real-world, you would contact a running service with a request for information over the network). This is why the line of code above, in that example execution controller, is executed once and the core compute speed is stored in the `core_flop_rate` variable to be re-used by the execution controller repeatedly throughout its execution.

2.5.3 B) Interacting with services

An execution controller can have many and complex interactions with services, especially with compute and storage services. In this section, we describe how WRENCH makes these interactions relatively easy, providing examples for each kind of interaction for each kind of service.

Job Manager and Data Movement Manager

As expected, each service type provides its own API. For instance, a network proximity service provides member functions to query the service's host distance databases. The *Developer API Reference* provides all necessary documentation, which also explains which member functions are synchronous and which are asynchronous (in which case some *event* will occur in the future). **However, the WRENCH developer will find that many member functions that one would expect are nowhere to be found. For instance, the compute services do not have (public) member functions for submitting jobs for execution!**

The rationale for the above is that many member functions need to be asynchronous so that the execution controller can use services concurrently. For instance, an execution controller could submit a job to two distinct compute services asynchronously, and then wait for the service which completes its job first and cancel the job on the other service. Exposing this asynchronicity to the execution controller would require that the WRENCH developer use data structures to perform the necessary bookkeeping of ongoing service interactions, and process incoming control messages from the services on the (simulated) network or alternately register many callbacks. Instead, WRENCH provides **managers**. One can think of managers as separate threads that handle all asynchronous interactions with services, and which have been implemented for your convenience to make interacting with services easy.

There are two managers: a **job manager** (class `wrench::JobManager`) and a **data movement manager** (class `wrench::DataMovementManager`). The base `wrench::ExecutionController` class provides two member functions for instantiating and starting these managers: `wrench::ExecutionController::createJobManager()` and `wrench::ExecutionController::createDataMovementManager()`.

Creating one or two of these managers typically is the first thing an execution controller does. For instance, the execution controller in `examples/workflow_api/basic-examples/bare-metal-data-movement/DataMovementWMS.cpp` starts by doing:

```
auto job_manager = this->createJobManager();
auto data_movement_manager = this->createDataMovementManager();
```

Each manager has its own documented API, and is discussed further in sections below.

Interacting with storage services

Typical interactions between an execution controller and a storage service include locating, reading, writing, and copying files. Different storage service implementations may or not implement some of of these operations. Click on the following links to see concrete examples of interactions with the currently available storage service type:

- Simple storage service
- XRootD storage service
- Storage service proxy

Interacting with compute services

The Job abstraction

The main activity of an execution controller is to execute workflow tasks on compute services. Rather than submitting tasks directly to compute services, an execution controller must create "jobs", which can comprise multiple tasks and involve data copy/deletion operations. The job abstraction is powerful and greatly simplifies the task of an execution controller while affording flexibility.

There are three kinds of jobs in WRENCH: `wrench::CompoundJob`, `wrench::StandardJob`, and `wrench::PilotJob`.

A **Compound Job** is simply set of actions to be performed, with possible control dependencies between actions. It is the most generic, flexible, and expressive kind of job. See the API documentation for the `wrench::CompoundJob` class and the examples in the `examples/action_api` directory. The other types of jobs below are actually implemented internally as compound jobs. The Compound Job abstraction is the most recent addition to the WRENCH API, and vastly expands the list of possible things that an execution controller can do. But because it is more recent, the reader will find that there are more examples in these documents and in the `examples` directory that use standard jobs (described below). But all these examples could be easily rewritten using the more generic compound job abstraction.

A **Standard Job** is a specific kind of job designed for **workflow** applications. In its most complete form, a standard job specifies:

- A set (in fact a vector) of `std::shared_ptr<wrench::WorkflowTask>` to execute, so that each task without all its predecessors in the set is ready;
- A `std::map` of `<std::shared_ptr<wrench::DataFile>>, std::shared_ptr<wrench::StorageService>>` pairs that specifies from which storage services particular input files should be read and to which storage services output files should be written;
- A set of file copy operations to be performed before executing the tasks;
- A set of file copy operations to be performed after executing the tasks; and
- A set of file deletion operations to be performed after executing the tasks and file copy operations.

Any of the above can actually be empty, and in the extreme a standard job can do nothing.

A **Pilot Job** (sometimes called a "placeholder job" in the literature) is a concept that is mostly relevant for batch scheduling. In a nutshell, it is a job that allows late binding of tasks to resources. It is submitted to a compute service (provided that service supports pilot jobs), and when it starts it just looks to the execution controller like a short-lived `wrench::BareMetalComputeService` to which compound and/or standard jobs can be submitted.

All jobs are created via the job manager, which provides `wrench::JobManager::createCompoundJob()`, `wrench::JobManager::createStandardJob()`, and `wrench::JobManager::createPilotJob()` member functions (the job manager is thus a job factory).

In addition to member functions for job creation, the job manager also provides the following:

- `wrench::JobManager::submitJob()`: asynchronous submission of a job to a compute service.
- `wrench::JobManager::terminateJob()`: synchronous termination of a previously submitted job.

The next section gives examples of interactions with each kind of compute service.

Click on the following links to see detailed descriptions and examples of how jobs are submitted to each compute service type:

- Bare-metal compute service
- Batch compute service
- Cloud compute service
- Virtualized cluster compute service
- HTCondor compute service

Interacting with file registry services

Interaction with a file registry service is straightforward and done by directly calling member functions of the `wrench::FileRegistryService` class. Note that often file registry service entries are managed automatically, e.g., via calls to `wrench::DataMovementManager` and `wrench::StorageService` member functions. So often an execution controller does not need to interact with the file registry service.

Adding/removing an entry to a file registry service is done as follows:

```
std::shared_ptr<wrench::FileRegistryService> file_registry;
std::shared_ptr<wrench::DataFile> some_file;
std::shared_ptr<wrench::StorageService> some_storage_service;

[...]

file_registry->addEntry(wrench::FileLocation::LOCATION(some_storage_service, some_file));
file_registry->removeEntry(wrench::FileLocation::LOCATION(some_storage_service, some_
↪file));
```

The `wrench::FileLocation` class is a convenient abstraction for a file that is available at some storage service (with optionally a directory path at that service).

Retrieving all entries for a given file is done as follows:

```
std::shared_ptr<wrench::FileRegistryService> file_registry;
std::shared_ptr<wrench::DataFile> some_file;

[...]
```

(continues on next page)

(continued from previous page)

```
std::set<std::shared_ptr<wrench::FileLocation>> entries;
entries = file_registry->lookupEntry(some_file);
```

If a network proximity service is running, it is possible to retrieve entries for a file sorted by non-decreasing proximity from some reference host. Returned entries are stored in a (sorted) `std::map` where the keys are network distances to the reference host. For instance:

```
std::shared_ptr<wrench::FileRegistryService> file_registry;
std::shared_ptr<wrench::DataFile> some_file;
std::shared_ptr<wrench::NetworkProximityService> np_service;

[...]

auto entries = fr_service->lookupEntry(some_file, "ReferenceHost", np_service);
```

See the documentation of `wrench::FileRegistryService` for more API member functions.

Interacting with network proximity services

Querying a network proximity service is straightforward. For instance, to obtain a measure of the network distance between hosts "Host1" and "Host2", one simply does:

```
std::shared_ptr<wrench::NetworkProximityService> np_service;

std::pair<double,double> distance = np_service->getHostPairDistance(std::make_pair("Host1
↪", "Host2"));
```

This distance corresponds to half the round-trip-time, in seconds, between the two hosts. The second value of the pair is the timestamp of the oldest measurement uses to compute the proximity value. If the service is configured to use the Vivaldi coordinate-based system, as in our example above, this distance is actually derived from network coordinates, as computed by the Vivaldi algorithm. In this case, one can actually ask for these coordinates for any given host:

```
std::pair<std::pair<double,double>, double> coords = np_service->getHostCoordinate("Host1
↪");
```

See the documentation of `wrench::NetworkProximityService` for more API member functions.

2.5.4 C) Workflow execution events

Because the execution controller performs asynchronous operations, it needs to wait for and re-act to events. This is done by calling the `wrench::ExecutionController::waitForAndProcessNextEvent()` member function implemented by the base `wrench::ExecutionController` class. A call to this member function blocks until some event occurs and then calls a callback member function. The possible event classes all derive from the `wrench::ExecutionEvent` class, and an execution controller can override the callback member function for each possible event (the default member function does nothing but print some log message). These overridable callback member functions are:

- `wrench::ExecutionController::processEventCompoundJobCompletion()`: react to a compound job completion
- `wrench::ExecutionController::processEventCompoundJobFailure()`: react to a compound job failure

- `wrench::ExecutionController::processEventStandardJobCompletion()`: react to a standard job completion
- `wrench::ExecutionController::processEventStandardJobFailure()`: react to a standard job failure
- `wrench::ExecutionController::processEventPilotJobStart()`: react to a pilot job beginning execution
- `wrench::ExecutionController::processEventPilotJobExpiration()`: react to a pilot job expiration
- `wrench::ExecutionController::processEventFileCopyCompletion()`: react to a file copy completion
- `wrench::ExecutionController::processEventFileCopyFailure()`: react to a file copy failure

Each member function above takes in an event object as parameter. In the case of failure, the event includes a `wrench::FailureCause` object, which can be accessed to analyze (or just display) the root cause of the failure.

Consider the execution controller in `examples/workflow_api/basic-examples/bare-metal-bag-of-tasks/TwoTasksAtATimeWMS.cpp`. At each each iteration of its main loop it does:

```
// Submit some standard job to some compute  service
job_manager->submitJob(...);

// Wait for and process next event
this->waitForAndProcessNextEvent();
```

In this simple example, only one of two events could occur at this point: a standard job completion or a standard job failure. As a result, this execution controller overrides the two corresponding member functions as follows:

```
void TwoTasksAtATimeWMS::processEventStandardJobCompletion(
                std::shared_ptr<StandardJobCompletedEvent> event) {
  // Retrieve the job that this event is for
  auto job = event->standard_job;
  // Print some message for each task in the job
  for (auto const &task : job->getTasks()) {
    std::cerr  << "Notified that a standard job has completed task " << task->getID() <<␣
↪std::endl;
  }
}

void TwoTasksAtATimeWMS::processEventStandardJobFailure(
                std::shared_ptr<StandardJobFailedEvent> event) {
  // Retrieve the job that this event is for
  auto job = event->standard_job;
  std::cerr  << "Notified that a standard job has failed (failure cause: ";
  std::cerr << event->failure_cause->toString() << ")" <<  std::endl;
  // Print some message for each task in the job if it has failed
  std::cerr << "As a result, the following tasks have failed:";
  for (auto const &task : job->getTasks()) {
    if (task->getState != WorkflowTask::COMPLETE) {
      std::cerr  << "  - " << task->getID() << std::endl;
    }
  }
}
```

You may note some difference between the above code and that in `examples/workflow_api/basic-examples/bare-metal-bag-of-tasks/TwoTasksAtATimeWMS.cpp`. This is for clarity purposes, and especially because we have not yet explained how WRENCH does message logging. See *an upcoming section about logging*.

While the above callbacks are convenient, sometimes it is desirable to do things more manually. That is, wait for an event and then process it in the code of the main loop of the execution controller rather than in a callback member function. This is done by calling the `wrench::waitForNextEvent()` member function. For instance, the execution controller in `examples/workflow_api/basic-examples/bare-metal-data-movement/DataMovementWMS.cpp` does it as:

```
// Initiate an asynchronous file copy
data_movement_manager->initiateAsynchronousFileCopy(...);

// Wait for an event
auto event = this->waitForNextEvent();

//Process the event
if (auto file_copy_completion_event = std::dynamic_pointer_cast
↪<wrench::FileCopyCompletedEvent>(event)) {
  std::cerr << "Notified of a file copy completion for file ";
  std::cerr << file_copy_completion_event->file->getID()<< "as expected" << std::endl;
} else {
   throw std::runtime_error("Unexpected event (" + event->toString() + ")");}
}
```

2.5.5 Exceptions

Most member functions in the WRENCH Developer API throw exceptions. In fact, most of the code fragments above should be in try-catch clauses, catching these exceptions.

Some exceptions correspond to failures during the simulated workflow executions (i.e., errors that would occur in a real-world execution and are thus part of the simulation). Each such exception contains a `wrench::FailureCause` object, which can be accessed to understand the root cause of the execution failure. Other exceptions (e.g., `std::invalid_arguments`, `std::runtime_error`) are thrown as well, which are used for detecting misuses of the WRENCH API or internal WRENCH errors.

2.5.6 Finding information and interacting with hardware resources

The `wrench::Simulation` class provides many member functions to discover information about the (simulated) hardware platform and interact with it. It also provides other useful information about the simulation itself, such as the current simulation date. Some of these member functions are static, but others are not. The `wrench::ExecutionController` class includes a `simulation` object. Thus, the execution controller can call member functions on the `this->simulation` object. For instance, this fragment of code shows how an execution controller can figure out the current simulated date and then check that a host exists (given a hostname) and, if so, set its `pstate` (power state) to the highest possible setting.

```
auto now = wrench::Simulation::getCurrentSimulatedDate();
if (wrench::Simulation::doesHostExist("SomeHost"))  {
  this->simulation->setPstate("SomeHost", wrench::Simulation::getNumberofPstates(
↪"SomeHost")-1);
}
```

See the documentation of the `wrench::Simulation` class for all details. Specifically regarding host pstates, see the example execution controller in `examples/workflow_api/basic-examples/cloud-bag-of-tasks-energy/TwoTasksAtATimeCloudWMS.cpp`, which interacts with host pstates (and the `examples/workflow_api/basic-examples/cloud-bag-of-tasks-energy/four_hosts_energy.xml` platform description file which defines pstates).

2.5.7 Logging

It is typically desirable for the execution controller to print log output to the terminal. This is easily accomplished using the `wrench::WRENCH_INFO()`, `wrench::WRENCH_DEBUG()`, and `wrench::WRENCH_WARN()` macros, which are used just like C's `printf()`. Each of these macros corresponds to a different logging level in SimGrid. See the SimGrid logging documentation for all details.

Furthermore, one can change the color of the log messages with the `wrench::TerminalOutput::setThisProcessLoggingColor()` member function, which takes as parameter a color specification:

- `wrench::TerminalOutput::COLOR_BLACK`
- `wrench::TerminalOutput::COLOR_RED`
- `wrench::TerminalOutput::COLOR_GREEN`
- `wrench::TerminalOutput::COLOR_YELLOW`
- `wrench::TerminalOutput::COLOR_BLUE`
- `wrench::TerminalOutput::COLOR_MAGENTA`
- `wrench::TerminalOutput::COLOR_CYAN`
- `wrench::TerminalOutput::COLOR_WHITE`

When inspecting the code of the execution controllers in the example simulators you will find many examples of calls to `wrench::WRENCH_INFO()`. The logging is per `.cpp` file, each of which corresponds to a declared logging category. For instance, in `examples/workflow_api/basic-examples/batch-bag-of-tasks/TwoTasksAtATimeBatchWMS.cpp`, you will find the typical pattern:

```
// Define a log category name for this file
WRENCH_LOG_CATEGORY(custom_wms, "Log category for TwoTasksAtATimeBatchWMS");

[...]

int TwoTasksAtATimeBatchWMS::main() {

  // Set the logging color to green
  TerminalOutput::setThisProcessLoggingColor(TerminalOutput::COLOR_GREEN);

  [...]

  // Print an info-level message, using printf-like format
  WRENCH_INFO("Submitting the job, asking for %s %s-core nodes for %s seconds",
              service_specific_arguments["-N"].c_str(),
              service_specific_arguments["-c"].c_str(),
              service_specific_arguments["-t"].c_str());

  [...]

  // Print a last info-level message
  WRENCH_INFO("Workflow execution complete");
  return 0;
}
```

The name of the logging category, in this case `custom_wms`, can then be passed to the `--log` command-line argument. For instance, invoking the simulator with additional argument `--log=custom_wms.threshold=info` will make it so that only those `WRENCH_INFO` statements in `TwoTasksAtATimeBatchWMS.cpp` will be printed (in green!).

2.6 WRENCH User API

Runtime System Users use WRENCH to simulate application workload executions using an already available, in-simulation implementation of a runtime system that uses Core Services to execution that workload.

Navigate through the sidebar to view the documentation for each class under the WRENCH User API.

2.7 WRENCH Developer API

Runtime System Developers/Researchers use WRENCH to prototype and evaluate runtime system designs and/or to investigate and evaluate novel algorithms to be implemented in a runtime system.

Navigate through the sidebar to view the documentation for each class under the WRENCH Developer API.

2.8 WRENCH Internal API

Internal Developers contribute to the WRENCH code, typically by implementing new Core Services.

Navigate through the sidebar to view the documentation for each class under the WRENCH Internal API.

2.9 WRENCH REST API

WRENCH provides a REST API, so as to provide a language-agnostic way to develop WRENCH simulators (at the cost of extra overhead). To this end, the WRENCH distribution comes with a "WRENCH daemon" executable, which can be built and installed as:

```
make wrench-daemon
make install  # try "sudo make install" if you do not have write privileges
```

The wrench-daemon is to be started on your local machine (see `wrench-daemon --help` for command-line options). and comprises an HTTP server that answers REST API requests.

The full documentation of the REST API is provided on this page

We have developed a Python API to WRENCH, which sits on top of the above REST API.