Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH

Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, Frédéric Suter

Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH

Henri Casanova[a,*], Rafael Ferreira da Silva[b,c,**], Ryan Tanaka[a,b], Suraj Pandey[a], Gautam Jethwani[c], William Koch[a], Spencer Albrecht[c], James Oeth[c], Frédéric Suter[d]

[a] *Information and Computer Sciences, University of Hawaii, Honolulu, HI, USA*
[b] *University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*
[c] *University of Southern California, Department of Computer Science, Los Angeles, CA, USA*
[d] *IN2P3 Computing Center, CNRS, Villeurbanne, France*

Abstract

Scientific workflows are used routinely in numerous scientific domains, and Workflow Management Systems (WMSs) have been developed to orchestrate and optimize workflow executions on distributed platforms. WMSs are complex software systems that interact with complex software infrastructures. Most WMS research and development activities rely on empirical experiments conducted with full-fledged software stacks on actual hardware platforms. These experiments, however, are limited to hardware and software infrastructures at hand and can be labor- and/or time-intensive. As a result, relying solely on real-world experiments impedes WMS research and development. An alternative is to conduct experiments in simulation. In this work we present WRENCH, a WMS simulation framework, whose objectives are (*i*) accurate and scalable simulations; and (*ii*) easy simulation software development. WRENCH achieves its first objective by building on the SimGrid framework. While SimGrid is recognized for the accuracy and scalability of its simulation models, it only provides low-level simulation abstractions and thus large software development efforts are required when implementing simulators of complex systems. WRENCH thus achieves its second objective by providing high-level and directly re-usable simulation abstractions on top of SimGrid. After describing and giving rationales for WRENCH's software architecture and APIs, we present two case studies in which we apply WRENCH to simulate the Pegasus production WMS and the WorkQueue application execution framework. We report on ease of implementation, simulation accuracy, and simulation scalability so as to determine to which extent WRENCH achieves its objectives. We also draw both qualitative and quantitative comparisons with a previously proposed workflow simulator.

Keywords: Scientific Workflows, Workflow Management Systems, Simulation, Distributed Computing

1. Introduction

Scientific workflows have become mainstream in support of research and development activities in numerous scientific domains [1]. Consequently, several Workflow Management Systems (WMSs) have been developed [2, 3, 4, 5, 6, 7] that allow scientists to execute workflows on distributed platforms that can accommodate executions at various scales. WMSs handle the logistics of workflow executions and make decisions regarding resource selection, data management, and computation scheduling, the goal being to optimize some performance metric (e.g., latency [8, 9], throughput [10, 11], jitter [12], reliability [13, 14, 15], power consumption [16, 17]). WMSs are complex software systems that interact with complex software infrastructures and can thus employ a wide range of designs and algorithms.

In spite of active WMS development and use in production, which has entailed solving engineering challenges, fundamental questions remain unanswered in terms of system designs and algorithms. Although there are theoretical underpinnings for most of these questions, theoretical results often make assumptions that do not hold with production hardware and software infrastructures. Further, the specifics of the design of a WMS can impose particular constraints on what solutions can be implemented effectively, and these constraints are typically not considered in available theoretical results. Consequently, current research that aims at improving and evolving the state of the art, although sometimes informed by theory, is mostly done via "real-world" experiments: designs and algorithms are implemented, evaluated, and selected based on experiments conducted for a particular WMS implementation with particular workflow configurations on particular platforms. As a corollary, from the WMS user's perspective, quantifying accurately how a WMS would perform for a particular workflow configuration on a particular platform entails actually executing that

*Corresponding address: University of Hawaii Information and Computer Sciences, POST Building, Rm 317, 1680 East-West Road, Honolulu, HI, USA, 96822

**Corresponding address: USC Information Sciences Institute, 4676 Admiralty Way Suite 1001, Marina del Rey, CA, USA, 90292

Email addresses: henric@hawaii.edu (Henri Casanova), rafsilva@isi.edu (Rafael Ferreira da Silva), tanaka@isi.edu (Ryan Tanaka), surajp@hawaii.edu (Suraj Pandey), gautam.jethwani@usc.edu (Gautam Jethwani), kochwill@hawaii.edu (William Koch), spencera@usc.edu (Spencer Albrecht), oeth@usc.edu (James Oeth), frederic.suter@cc.in2p3.fr (Frédéric Suter)

workflow on that platform.

Unfortunately, real-world experiments have limited scope, which impedes WMS research and development. This is because they are confined to application and platform configurations available at hand, and thus cover only a small subset of the relevant scenarios that may be encountered in practice. Furthermore, exclusively relying on real-world experiments makes it difficult or even impossible to investigate hypothetical scenarios (e.g., "What if the network had a different topology?", "What if there were 10 times more compute nodes but they had half as many cores?"). Real-world experiments, especially when large-scale, are often not fully reproducible due to shared networks and compute resources, and due to transient or idiosyncratic behaviors (maintenance schedules, software upgrades, and particular software (mis)configurations). Running real-world experiments is also time-consuming, thus possibly making it difficult to obtain statistically significant numbers of experimental results. Real-world experiments are driven by WMS implementations that often impose constraints on workflow executions. Furthermore, WMSs are typically not monolithic but instead reuse CyberInfrastructure (CI) components that impose their own overheads and constraints on workflow execution. Exploring what lies beyond these constraints via real-world executions, e.g., for research and development purposes, typically entails unacceptable software (re-)engineering costs. Finally, running real-world experiments can also be labor-intensive. This is due to the need to install and execute many full-featured software stacks, including actual scientific workflow implementations, which is often not deemed worthwhile for "just testing out" ideas.

An alternative to conducting WMS research via real-world experiments is to use *simulation*, i.e., implement a software artifact that models the functional and performance behaviors of software and hardware stacks of interest. Simulation is used in many computer science domains and can address the limitations of real-world experiments outlined above. Several simulation frameworks have been developed that target the parallel and distributed computing domain [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]. Some simulation frameworks have also been developed specifically for the scientific workflow domain [35, 36, 37, 11, 38, 39, 40].

We claim that advances in simulation capabilities in the field have made it possible to simulate WMSs that execute large workflows using diverse CI services deployed on large-scale platforms in a way that is accurate (via validated simulation models), scalable (fast execution and low memory footprint), and expressive (ability to describe arbitrary platforms, complex WMSs, and complex software infrastructure). In this work, we build on the existing open-source SimGrid simulation framework [33, 41], which has been one of the drivers of the above advances and whose simulation models have been extensively validated [42, 43, 44, 45, 46], to develop a WMS simulation framework called WRENCH [47]. More specifically, this work makes the following contributions[1]:

1. We justify the need for WRENCH and explain how it improves on the state of the art.

2. We describe the high-level simulation abstractions provided by WRENCH that (i) make it straightforward to implement full-fledged simulated versions of complex WMS systems; and (ii) make it possible to instantiate simulation scenarios with only few lines of code.

3. Via two case studies with the Pegasus [2] production WMS and the WorkQueue [49] application execution framework, we evaluate the ease-of-use, accuracy, and scalability of WRENCH, and compare it with a previously proposed simulator, WorkflowSim [35].

This paper is organized as follows. Section 2 discusses related work. Section 3 outlines the design of WRENCH and describes how its APIs are used to implement simulators. Section 4 presents our case studies. Finally, Section 5 concludes with a brief summary of results and a discussion of future research directions.

2. Related Work

Many simulation frameworks have been developed for parallel and distributed computing research and development. They span domains such as HPC [18, 19, 20, 21], Grid [22, 23, 24], Cloud [25, 26, 27], Peer-to-peer [28, 29], or Volunteer Computing [30, 31, 32]. Some frameworks have striven to be applicable across some or all or the above domains [33, 34]. Two conflicting concerns are *accuracy* (the ability to capture the behavior of a real-world system with as little bias as possible) and *scalability* (the ability to simulate large systems with as few CPU cycles and bytes of RAM as possible). The aforementioned simulation frameworks achieve different compromises between these two concerns by using various simulation models. At one extreme are discrete event models that simulate the "microscopic" behavior of hardware/software systems (e.g., by relying on packet-level network simulation for communication [50], on cycle-accurate CPU simulation [51] or emulation for computation). In this case, the scalability challenge can be handled by using Parallel Discrete Event Simulation [52], i.e., the simulation itself is a parallel application that requires a parallel platform whose scale is at least commensurate to that of the simulated platform. At the other extreme are analytical models that capture "macroscopic" behaviors (e.g., transfer times as data sizes divided by bottleneck bandwidths, compute times as numbers of operations divided by compute speeds). While these models are typically more scalable, they must be developed with care so that they are accurate. In previous work, it has been shown that several available simulation frameworks use macroscopic models that can exhibit high inaccuracy [43].

[1] A preliminary shorter version of this paper appears in the proceedings of the 2018 Workshop on Workflows in Support of Large-Scale Science (WORKS) [48].

A number of simulators have been developed that target scientific workflows. Some of them are stand-alone simulators [35, 36, 37, 11, 53]. Others are integrated with a particular WMS to promote more faithful simulation and code reuse [38, 39, 54] or to execute simulations at runtime to guide on-line scheduling decisions made by the WMS [40].

The authors in [39] conduct a critical analysis of the state-of-the-art of workflow simulators. They observe that many of these simulators do not capture the details of underlying infrastructures and/or use naive simulation models. This is the case with custom simulators such as that in [40, 36, 37]. But it is also the case with workflow simulators built on top of generic simulation frameworks that provide convenient user-level abstractions but fail to model the details of the underlying infrastructure, e.g., the simulators in [35, 38, 11], which build on the CloudSim [25] or GroudSim [24] frameworks. These frameworks have been shown to lack in their network modeling capabilities [43]. As a result, some authors readily recognize that their simulators are likely only valid when network effects play a small role in workflow executions (i.e., when workflows are not data-intensive).

To overcome the above limitations, in [39, 54] the authors have improved the network model in GroudSim and also use a separate simulator, DISSECT-CF [27], for simulating cloud infrastructures accurately. The authors acknowledge that the popular SimGrid [33, 41] simulation framework offers compelling capabilities, both in terms of scalability and simulation accuracy. But one of their reasons for not considering SimGrid is that, because it is low-level, using it to implement a simulator of a complex system, such as a WMS and the CI services it uses, would be too labor-intensive. In this work, we address this issue by developing a simulation framework that provides convenient, reusable, high-level abstractions but that builds on SimGrid so as to benefit from its scalable and accurate simulation models. Furthermore, unlike [38, 39, 54], we do not focus on integration with any specific WMS. The argument in [39] is that stand-alone simulators, such as that in [35], are disconnected from real-world WMSs because they abstract away much of the complexity of these systems. Instead, our proposed framework does capture low-level system details (and simulates them well thanks to SimGrid), but provides high-level enough abstractions to implement faithful simulations of complex WMSs with minimum effort, which we demonstrate via two case studies.

Also related to this work is previous research that has not focused on providing simulators or simulation frameworks *per se*, but instead on WMS simulation methodology. In particular, several authors have investigated methods for injecting realistic stochastic noise in simulated WMS executions [35, 55]. These techniques can be adopted by most of the aforementioned frameworks, including the one proposed in this work.

3. WRENCH

3.1. Objective and Intended Users

WRENCH's objective is to make it possible to study WMSs in simulation in a way that is accurate (faithful modeling of real-world executions), scalable (low computation and memory footprints on a single computer), and expressive (ability to simulate arbitrary WMS, workflow, and platform scenarios with minimal software engineering effort). WRENCH is not a simulator but a simulation framework that is distributed as a C++ library. It provides high-level reusable abstractions for developing simulated WMS implementations and simulators for the execution of these implementations. There are two categories of WRENCH users:

1. *Users who implement simulated WMSs* – These users are engaged in WMS research and development activities and need an "in simulation" version of their current or intended WMS. Their goals typically include evaluating how their WMS behaves over hypothetical experimental scenarios and comparing competing algorithm and system design options. For these users, WRENCH provides the WRENCH Developer API (described in Section 3.4) that eases WMS development by removing the typical difficulties involved when developing, either in real-world or in simulation mode, a system comprised of distributed components that interact both synchronously and asynchronously. To this end, WRENCH makes it possible to implement a WMS as a single thread of control that interacts with simulated CI services via high-level APIs and must react to a small set of asynchronous events.
2. *Users who execute simulated WMSs* – These users simulate how given WMSs behave for particular workflows on particular platforms. Their goals include comparing different WMSs, determining how a given WMS would behave for various workflow configurations, comparing different platform and resource provisioning options, determining performance bottlenecks, engaging in pedagogic activities centered on distributed computing and workflow issues, etc. These users can develop simulators via the WRENCH User API (described in Section 3.5), which makes it possible to build a full-fledged simulator with only a few lines of code.

Users in the first category above often also belong to the second category. That is, after implementing a simulated WMS these users typically instantiate simulators for several experimental scenarios to evaluate their WMS.

3.2. Software Architecture Overview

Figure 1 depicts WRENCH's software architecture. At the bottom layer is the Simulation Core, which simulates low-level software and hardware stacks using the simulation abstractions and models provided by SimGrid (see Section 3.3). The next layer implements simulated CI services that are commonly found in current distributed platforms and used by production WMSs. At the time of this writing, WRENCH provides services in 4 categories: *compute services* that provide access to compute resources to execute workflow tasks; *storage services* that provide access to storage resources for storing workflow data; *network monitoring services* that can be queried to determine network distances; and *data registry services* that can be used to track the location of (replicas of) workflow data. Each category includes multiple service implemen-

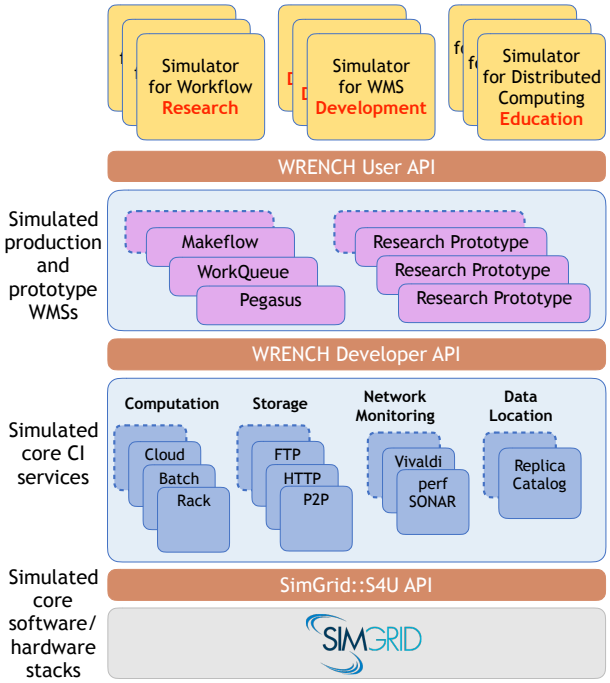

Figure 1: The four layers in the WRENCH architecture from bottom to top: simulation core, simulated core services, simulated WMS implementations, and simulators.

tations, so as to capture specifics of currently available CI services used in production. For instance, WRENCH includes a "batch-scheduled cluster" compute service, a "cloud" compute service, and a "bare-metal" compute service. The above layer in the software architecture consists of simulated WMS, that interact with CI services using the WRENCH Developer API (see Section 3.4). These WMS implementations, which can simulate production WMSs or WMS research prototypes, are not included as part of the WRENCH distribution, but implemented as stand-alone projects. Two such projects are the simulated Pegasus and Workqueue implementations used for our case study in Section 4. Finally, the top layer consists of simulators that configure and instantiate particular CI services and particular WMSs on a given simulated hardware platform, that launch the simulation, and that analyze the simulation outcome. These simulators use the WRENCH User API (see Section 3.5). Here again, these simulators are not part of WRENCH, but implemented as stand-alone projects.

3.3. Simulation Core

WRENCH's simulation core is implemented using SimGrid's S4U API, which provides all necessary abstractions and models to simulate computation, I/O, and communication activities on arbitrary hardware platform configurations. These platform configurations are defined by XML files that specify network topologies and endpoints, compute resources, and storage resources [56].

At its most fundamental level, SimGrid provides a Concurrent Sequential Processes (CSP) model: a simulation consists of sequential threads of control that consume hardware resources.

Algorithm 1 Blueprint for a WMS execution

```
1:  procedure MAIN(workflow)
2:      Obtain list of available services
3:      Gather static information about the services
4:      while workflow execution has not completed/failed do
5:          Gather dynamic service/resource information
6:          Make data/computation scheduling decisions
7:          Interact with services to enact decisions
8:          Wait for and react to the next event
9:      end while
10:     return
11: end procedure
```

These threads of control can implement arbitrary code, exchange messages via a simulated network, can perform computation on simulated (multicore) hosts, and can perform I/O on simulated storage devices. In addition, SimGrid provides a virtual machine abstraction that includes a migration feature. Therefore, SimGrid provides all the base abstractions necessary to implement the classes of distributed systems that are relevant to scientific workflow executions. However, these abstractions are low-level and a common criticism of SimGrid is that implementing a simulation of a complex system requires a large software engineering effort. A WMS executing a workflow using several CI services is a complex system, and WRENCH builds on top of SimGrid to provide high-level abstractions so that implementing this complex system is not labor-intensive.

We have selected SimGrid for WRENCH for the following reasons. SimGrid has been used successfully in many distributed computing domains (cluster, peer-to-peer, grid, cloud, volunteer computing, etc.), and thus can be used to simulate WMSs that execute over a wide range of platforms. SimGrid is open source and freely available, has been stable for many years, is actively developed, has a sizable user community, and has provided simulation results for over 350 research publications since its inception. SimGrid has also been the object of many invalidation and validation studies [42, 43, 44, 45, 46], and its simulation models have been shown to provide compelling advantages over other simulation frameworks in terms of both accuracy and scalability [33]. Finally, most SimGrid simulations can be executed in minutes on a standard laptop computer, making it possible to perform large numbers of simulations quickly with minimal compute resource expenses. To the best of our knowledge, among comparable available simulation frameworks (as reviewed in Section 2), SimGrid is the only one to offer all the above desirable characteristics.

3.4. WRENCH Developer API

With the Developer API, a WMS is implemented as a single thread of control that executes according to the pseudo-code blueprint shown in Algorithm 1. Given a workflow to execute, a WMS first gathers information about all the CI services it can use to execute the workflow (lines 2-3). Examples of such information include the number of compute nodes provided by a compute service, the number of cores per node and the speed of these cores, the amount of storage space available in a storage

service, the list of hosts monitored by a network monitoring service, etc. Then, the WMS iterates until the workflow execution is complete or has failed (line 4). At each iteration it gathers dynamic information about available services and resources if needed (line 5). Example of such information include currently available capacities at compute or storage services, current network distances between pairs of hosts, etc. Based on resource information and on the current state of the workflow, the WMS can then make whatever scheduling decisions it sees fit (line 7). It then enacts these decisions by interacting with appropriate services. For instance, it could decide to submit a "job" to a compute service to execute a ready task on some number of cores at some compute service and copy all produced files to some storage service, or it could decide to just copy a file between storage services and then update a data location service to keep track of the location of this new file replica. It could also submit one or more *pilot jobs* [57] to compute services if they support them. It is the responsibility of the developer to implement all decision-making algorithms employed by the WMS. At the end of the iteration, the WMS simply waits for a workflow execution event to which it can react if need be. Most common events are job completions/failures and data transfer completions/failures.

The WRENCH Developer API provides a rich set of methods to create and analyze a workflow and to interact with CI services to execute a workflow. These methods were designed based on current and envisioned capabilities of current state-of-the-art WMSs. We refer the reader to the WRENCH Web site [47] for more information on how to use this API and for the full API documentation. The key objective of this API is to make it straightforward to implement a complex system, namely a full-fledged WMS that interact with diverse CI services. We achieve this objective by providing simple solutions and abstractions to handle well-known challenges when implementing a complex distributed system (whether in the real world or in simulation), as explained hereafter.

SimGrid provides simple point-to-point communication between threads of control via a mailbox abstraction. One of the recognized strengths of SimGrid is that it employs highly accurate and yet scalable network simulation models. However, unlike some of its competitors, it does not provide any higher-level simulation abstractions meaning that distributed systems must be implemented essentially from scratch, with message-based interactions between processes. All message-based interaction is abstracted away by WRENCH, and although the simulated CI services exchange many messages with the WMS and among themselves, the WRENCH Developer API only exposes higher-level interaction with services ("run this job", "move this data") and only requires that the WMS handle a few events. The WMS developer thus completely avoids the need to send and receive (and thus orchestrate) network messages.

Another challenge when developing a system like a WMS is the need to handle asynchronous interactions. While some service interactions can be synchronous (e.g., "are you up?", "tell me your current load"), most need to be asynchronous so that the WMS retains control. The typical solution is to maintain sets of request handles and/or to use multiple threads of control. To free the WMS developer from these responsibilities, WRENCH provides already implemented "managers" that can be used out-of-the-box to take care of asynchronicity. A WMS can instantiate such managers, which are independent threads of control. Each manager transparently interacts with CI services, maintains a set of pending requests, provides a simple API to check on the status of these requests, and automatically generates high-level workflow execution events. For instance, a WMS can instantiate a "job manager" through which it will create and submit jobs to compute services. It can at any time check on the status of a job, and the job manager interacts directly (and asynchronously) with compute services so as to generate "job done" or "job failed" events to which the WMS can react. In our experience developing simulators from scratch using SimGrid, the implementation of asynchronous interactions with simulated processes is a non-trivial development effort, both in terms of amount of code to write and difficulty to write this code correctly. We posit that this is one of the reasons why some users have preferred using simulation frameworks that provide higher-level abstractions than SimGrid even though they offer less attractive accuracy and/or scalability features. WRENCH provides such higher-level abstractions to the WMS developers, and as a result implementing a WMS with WRENCH can be straightforward.

Finally, one of the challenges when developing a WMS is failure handling. It is expected that compute, storage, and network resources, as well as the CI services that use them, can fail through the execution of the WMS. SimGrid has the capability to simulate arbitrary failures via availability traces. Furthermore, failures can occur due to the WMS implementation itself, e.g., if it fails to check that the operations it attempts are actually valid, if concurrent operations initiated by the WMS work at cross purposes. WRENCH abstracts away all these failures as C++ exceptions that can be caught by the WMS implementation, or caught by a manager and passed to the WMS as workflow execution events. Regardless, each failure exposes a failure cause, which encodes a detailed description of the failure. For instance, after initiating a file copy from a storage service to another storage service, a "file copy failed" event sent to the WMS would include a failure cause that could specify that when trying to copy file x from storage service y to storage service z, storage service z did not have sufficient storage space. Other example failure causes could be that a network error occurred when storage service y attempted to receive a message from storage service z, or that service z was down. All CI services implemented in WRENCH simulate well-defined failure behaviors, and failure handling capabilities afforded to simulated WMSs can actually allow more sophisticated failure tolerance strategies than currently done or possible in real-world implementations. But more importantly, the amount of code that needs to be written for failure handling in a simulated WMS is minimal.

Given the above, WRENCH makes it possible to implement a simulated WMS with very little code and effort. The example WMS implementation provided with the WRENCH distribution, which is simple but functional, is under 200 lines of C++ (once comments have been removed). See more discussion of

the effort needed to implement a WMS with WRENCH in the context of our case studies (Section 4).

3.5. WRENCH User API

With the User API one can quickly build a simulator, which typically follows these steps:

1. Instantiate a platform based on a SimGrid XML platform description file;
2. Create one or more workflows;
3. Instantiate services on the platform;
4. Instantiate one or more WMSs telling each what services are at its disposal and what workflow it should execute starting at what time;
5. Launch the simulation; and
6. Process the simulation outcome.

The above steps can be implemented with only a few lines of C++. An example is provided and described in Appendix A. This example showcases only the most fundamental features of the WRENCH User API, and we refer the reader to the WRENCH Web site [47] for more detailed information on how to use this API and for the full API documentation. In the future this API will come with Python binding so that users can implement simulators in Python.

3.6. Simulation Debugging and Visualization

Analyzing and interpreting simulation logs is often a labor-intensive process, and users of simulators typically develop sets of scripts for parsing and extracting specific knowledge from the logs. In order to provide WRENCH users with a rapid, first insight on their simulation results, we have been developing a Web-based "dashboard" that compiles simulation logs into a set of tabular and graphical JavaScript components. The dashboard presents overall views on task life-cycles, showing breakdowns between compute and I/O operations, as well as a Gantt chart and 2- and 3-dimensional plots of task executions and resource usage during the simulated workflow execution. An overview of energy consumption per compute resource can also be visualized in the dashboard. The dashboard is currently under development and will be available for users in the next WRENCH release. Figure 2 shows a screenshot of a simple simulated workflow execution.

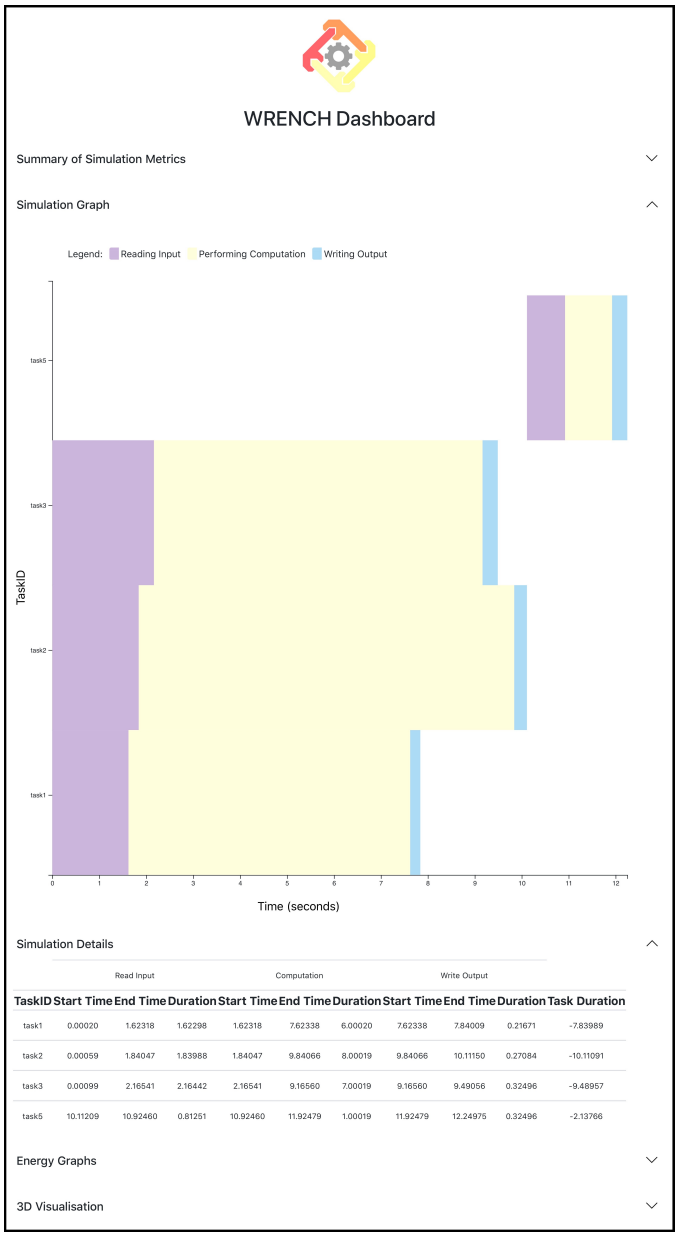

Figure 2: Screenshot of the Web-based WRENCH dashboard that shows, among other information not displayed here, execution details of each workflow task and an interactive Gantt chart of the task executions.

4. Case Study: Simulating production WMSs

In this section, we present two WRENCH-based simulators of a state-of-the-art WMS, Pegasus [2], and an application execution framework, WorkQueue [49], as case studies for evaluation and validation purposes.

Pegasus is being used in production to execute workflows for dozens of high-profile applications in a wide range of scientific domains [2]. Pegasus provides the necessary abstractions for scientists to create workflows and allows for transparent execution of these workflows on a range of compute platforms including clusters, clouds, and national cyberinfrastructures. During execution, Pegasus translates an abstract resource-independent workflow into an executable workflow, determining the specific executables, data, and computational resources required for the execution. Workflow execution with Pegasus includes data management, monitoring, and failure handling, and is managed by HTCondor DAGMan [58]. Individual workflow tasks are managed by a workload management framework, HTCondor [59], which supervises task executions on local and remote resources. Workflow executions with Pegasus follow a "push model," i.e., a task is bound to a particular compute resource at the onset of the execution and is always executed on that resource if possible.

WorkQueue is being used in production by a wide range of researchers across many scientific domains for building large-

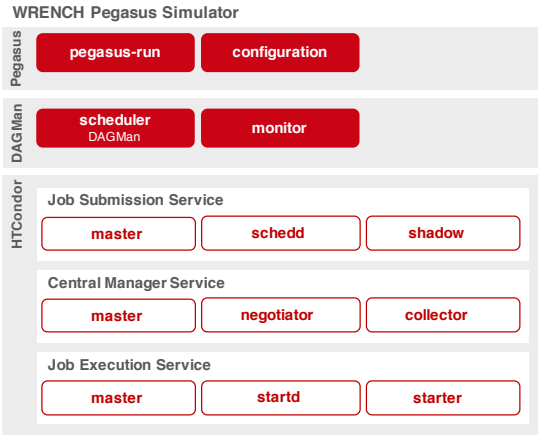

Figure 3: Overview of the WRENCH Pegasus simulation components, including components for DAGMan and HTCondor frameworks. Red boxes denote Pegasus services developed with WRENCH's Developer API, and white boxes denote WRENCH reused components.

scale master-worker applications that span thousands of machines drawn from clusters, clouds, and grids [60]. WorkQueue allows users to define tasks, submit them to a workqueue abstraction, and wait for their completions. During execution, WorkQueue starts standard worker processes that can run on any available compute resource. These worker processes perform data transfers and execute tasks, making it possible to execute workflows. Worker processes can also be submitted for execution on HTCondor pools, which is the approach evaluated in this paper. Workflow executions with WorkQueue follow a "pull model," i.e., late-binding of tasks to compute resources based on when resources becomes idle.

4.1. Implementing Pegasus with WRENCH

Since Pegasus relies on HTCondor, first we have implemented the HTCondor services as simulated core CI services, which together form a new Compute Service that exposes the WRENCH Developer API. This makes HTCondor available to any WMS implementation that is to be simulated using WRENCH, and has been included as part of the growing set of simulated core CI services provided by WRENCH.

HTCondor is composed of six main service daemons (`startd`, `starter`, `schedd`, `shadow`, `negotiator`, and `collector`). In addition, each host on which one or more of these daemons is spawned must also run a `master` daemon, which controls the execution of all other daemons (including initialization and completion). The bottom part of Figure 3 depicts the components of our simulated HTCondor implementation, where daemons are shown in red-bordered boxes. In our simulator we implement the 3 fundamental HTCondor services, implemented as particular sets of daemons, as depicted in the bottom part of Figure 3 in borderless white boxes. The *Job Execution Service* consists of a `startd` daemon, which adds the host on which it is running to the HTCondor pool, and of a `starter` daemon, which manages task executions on this host. The *Central Manager Service* consists of a `collector` daemon, which collects information about all other daemons, and of a `negotiator` daemon, which performs task/resource matchmaking. The *Job Submission Service* consists of a `schedd` daemon, which maintains a queue of tasks, and of several instances of a `shadow` daemon, each of which corresponds to a task submitted to the HTCondor pool for execution.

Given the simulated HTCondor implementation above, we then implemented the simulated Pegasus WMS, including the DAGMan workflow engine, using the WRENCH Developer API. This implementation instantiates all services and parses the workflow description file, the platform description file, and a Pegasus-specific configuration file. DAGMan orchestrates the workflow execution (e.g., a task is marked as ready for execution once all its parent tasks have successfully completed), and monitors the status of tasks submitted to the HTCondor pool using a pull model, i.e., task status is fetched from the pool at regular time intervals. The top part of Figure 3 depicts the components of our simulated Pegasus implementation (each shown in a red box).

By leveraging WRENCH's high-level simulation abstractions, implementing HTCondor as a reusable core WRENCH service using the Developer API required only 613 lines of code. Similarly, implementing a simulated version of Pegasus, including DAGMan, was done with only 666 lines of code (127 of which are merely parsing simulation configuration files). These numbers include both header and source files, but exclude comments. We argue that the above corresponds to minor simulation software development efforts when considering the complexity of the system being simulated.

Service implementations in WRENCH are all parameterizable. For instance, as services use message-based communications it is possible to specify all message payloads in bytes (e.g., for control messages). Other parameters encompass various overheads, either in seconds or in computation volumes (e.g., task startup overhead on a compute service). In WRENCH, service implementations come with default values for all these parameters, but it is possible to pick custom values upon service instantiation. The process of picking parameter values so as to match a specific real-world system is referred to as *simulation calibration*. We calibrated our simulator by measuring delays observed in event traces of real-world executions for workflows on hardware/software infrastructures (see Section 4.3).

The simulator code, details on the simulation calibration procedure, and experimental scenarios used in the rest of this section are all publicly available online [61].

4.2. Implementing WorkQueue with WRENCH

WorkQueue implements a master-worker paradigm by which running worker processes can be assigned work dynamically. In our simulator, we implement workers as pilot jobs, a popular mechanism for late-binding of computation to resources [57], which is implemented in WRENCH. As WorkQueue does not provide automated mechanisms for starting workers, i.e., triggering pilot job submissions, at runtime, our simulated implementation limits the number of concurrently running pilot jobs based on available compute resources (i.e., cores are not oversubscribed). The simulator implementation has two main components: (1) a *workflow system* for orchestrating tasks execution and assigning compute tasks to workers; and (2) a *scheduler*

for submitting pilot jobs to compute services. As WRENCH provides a consistent interface across all compute services, our simulator can simulate the execution of WorkQueue on any compute service provided they are configured to support pilot job execution.

Due to WRENCH's providing high-level simulation abstractions, implementing WorkQueue using the WRENCH Developer API requires only 485 lines of code (113 of which are merely parsing simulation configuration files). These numbers include both header and source files, but exclude comments. Similarly to the WRENCH-based Pegasus simulator, we argue that the simulation software development effort is minimal when considering the complexity of the system being simulated. The simulator code, details on the simulation calibration procedure, and experimental scenarios used in the rest of this section are all publicly available online [62].

4.3. Experimental Scenarios

We consider experimental scenarios defined by particular workflow instances to be executed on particular platforms. Due to the lack of publicly available detailed workflow execution traces (i.e., execution logs that include data sizes for all files, all execution delays, etc.), we have performed real workflow executions with Pegasus and WorkQueue, and collected raw, time-stamped event traces from these executions. These traces form the ground truth to which we can compare simulated executions. We consider these workflow applications:

- *1000Genome* [63]: A data-intensive workflow that identifies mutational overlaps using data from the 1000 genomes project in order to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. We consider a 1000Genome instance that comprises 71 tasks.
- *Montage* [2]: A compute-intensive astronomy workflow for generating custom mosaics of the sky. For this experiment, we ran Montage for processing 1.5 and 2.0 square degrees mosaic 2MASS. We thus refer to each configuration as Montage-1.5 and Montage-2.0, respectively. Montage-1.5, resp. Montage-2.0, comprises 573, resp. 1,240, tasks.
- *SAND* [64]: A compute-intensive bioinformatics workflow for accelerating genome assembly. For this experiment, we ran SAND for a full set of reads from the Anopheles gambiae Mopti form. We consider a SAND instance that comprises 606 tasks.

We use these platforms, deploying on each a submit node (which runs Pegasus and DAGMan or WorkQueue, and HTCondor's job submission and central manager services), four worker nodes (4 or 24 cores per node / shared file system), and a data node in the WAN:

- *ExoGENI*: A widely distributed networked infrastructure-as-a-service testbed representative of a "bare metal" platform. Each worker node is a 4-core 2.0GHz processor with 12GiB of RAM. The bandwidth between the data node and the submit node was ~0.40 Gbps, and the bandwidth between the submit and worker nodes was ~1.00 Gbps.
- *AWS*: Amazon's cloud platform, on which we use two types of virtual machine instances: `t2.xlarge` and `m5.xlarge`. The bandwidth between the data node and the submit node was ~0.44 Gbps, and the bandwidth between the submit and worker nodes on these instances were ~0.74 Gbps and ~1.24 Gbps, respectively.
- *Chameleon*: An academic cloud testbed, on which we use homogeneous standard cloud units to run an HTCondor pool. Each unit consists of a 24-core 2.3GHz processor with 128 GiB of RAM. The bandwidth between the submit node and worker nodes on these instances were ~10.00 Gbps.

To evaluate the accuracy of our simulators, we consider 4 particular experimental scenarios: 1000Genome on ExoGENI, Montage-1.5 on AWS-t2.xlarge, Montage-2.0 on AWS-m5.xlarge, and SAND on Chameleon. The first three scenarios are performed using Pegasus, and the last one is performed using WorkQueue. For each scenario we repeat the real-world workflow execution 5 times (since real-world executions are not perfectly deterministic), and also run a simulated execution execution using the WRENCH-based simulators described in the previous section. For each execution, real-world or simulated, we keep track of the overall application makespan, but also of time-stamped individual execution events such as task submission and completions dates.

4.4. Pegasus: Simulation Accuracy

The fourth column in Table 1 shows average relative differences between actual and simulated makespans. We see that simulated makespans are close to actual makespans for all three Pegasus scenarios (average relative error is below 5%). One of the key advantages of building WRENCH on top of SimGrid is that WRENCH simulators benefit from the high-accuracy network models in SimGrid. In particular, these models capture many features of the TCP protocol (without resorting to packet-level simulation). And indeed, when comparing real-world and simulated executions we observe average relative error below 3% for data movement operations. Furthermore, the many processes involved in a workflow execution interact by exchanging (typically small) control messages, and our simulators simulate these message exchanges. For instance, each time an output file is produced by a task a data registry service is contacted so that a new entry can be added to its database of file replicas, which incurs some communication overhead. When comparing real-world to simulated executions we observe average relative simulation error below 1% for these data registration overheads. Overall, a key reason for the high accuracy of our simulators is that they simulate data and control message transfers accurately.

To draw comparisons with a state-of-the-art simulator, we repeated the above Pegasus simulation experiments using WorkflowSim [35]. WorkflowSim simulates workflow executions based on execution models similar to that of Pegasus and is built on top of the CloudSim simulation framework [25]. However, WorkflowSim does not provide a detailed simulated HTCondor implementation, and does not offer the same simulation calibration capabilities as WRENCH. Nevertheless, we have painstakingly calibrated the WorkflowSim simulator so that it

Workflow	Experimental Scenario System	Platform	Avg. Makespan Error (%)	Task Submissions *p*-value	Task Submissions distance	Tasks completions *p*-value	Tasks completions distance
1000Genome	Pegasus	ExoGENI	1.10 ±0.28	0.06 ±0.01	0.21 ±0.04	0.72 ±0.06	0.12 ±0.01
Montage-1.5	Pegasus	AWS-t2.xlarge	4.25 ±1.16	0.08 ±0.01	0.16 ±0.03	0.12 ±0.05	0.21 ±0.02
Montage-2.0	Pegasus	AWS-m5.xlarge	3.37 ±0.46	0.11 ±0.03	0.06 ±0.02	0.10 ±0.01	0.11 ±0.01
SAND	WorkQueue	Chameleon	3.96 ±1.04	0.06 ±0.01	0.11 ±0.02	0.09 ±0.02	0.09 ±0.03

Table 1: Average simulated makespan error (%), and *p*-values and Kolmogorov-Smirnov (KS) distances for task submission and completion dates, computed for 5 runs of each of our 4 experimental scenarios.

models the hardware and software infrastructures of our experimental scenarios as closely as possible. For each of the 3 Pegasus experimental scenarios, we find that the relative average makespan percentage error is 12.09 ±2.84, 26.87 ±6.26, and 13.32 ±1.12, respectively, i.e., from 4x up to 11x larger than the error values obtained with our WRENCH-based simulator. The reasons for the discrepancies between WorkflowSim and real-world results are twofold. First, WorkflowSim uses the simplistic network models in CloudSim (see discussion in Section 2) and thus suffers from simulation bias w.r.t. communication times. Second, WorkflowSim does not capture all the relevant system details of the system and of the workflow execution. By contrast, our WRENCH-based simulator benefits from the accurate network simulation models provided by SimGrid, and it does capture low-level relevant details although implemented with only a few hundred lines of code.

In our experiments, we also record the submission and completion dates of each task, thus obtaining empirical cumulative density functions (ECDFs) of these times, for both real-world executions and simulated executions. To further validate the accuracy of our simulation results, we apply Kolmogorov-Smirnov goodness of fit tests (KS tests) with null hypotheses (H_0) that the real-world and simulation samples are drawn from the same distributions. The two-sample KS test results in a miss if the null hypothesis (two-sided alternative hypothesis) is rejected at 5% significance level (p-value ≤ 0.05). Each test for which the null hypothesis is not rejected (p-value > 0.05), indicates that the simulated execution statistically matches the real-world execution. Table 1 shows p-value and KS test distance for both task submission times and task completion times. The null hypothesis is not rejected, and we thus conclude that simulated workflow task executions statistically match real-world executions well. These conclusions are confirmed by visually comparing ECDFs. For instance, Figure 4 shows real-world and simulated ECDFs for sample runs of Montage-2.0 on AWS-m5.xlarge, with task submission, resp. completion, date ECDFs on the left-hand, resp. right-hand, side. We observe that the simulated ECDFs ("wrench") track the real-world ECDFs ("pegasus") closely. We repeated these simulations using WorkflowSim, and found that the null hypothesis is rejected for all 3 simulation scenarios. This is confirmed visually in Figure 4, where the ECDFs obtained from the WorkflowSim simulation ("workflowsim") are far from the real-world ECDFs.

Although KS tests and ECDFs visual inspections validate that the WRENCH-simulated ECDFs match the real-world ECDFs statistically, these results do not distinguish between individual tasks. In fact, there are some discrepancies between

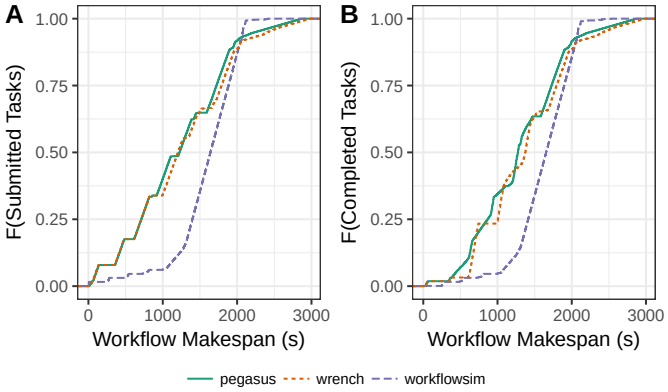

Figure 4: Empirical cumulative distribution function of task submit times (left) and task completion times (right) for sample real-world ("pegasus") and simulated ("wrench" and "workflowsim") executions of Montage-2.0 on AWS-m5.xlarge.

real-world and simulated schedules. For instance, Figure 5 shows Gantt charts corresponding to the workflow executions shown in Figure 4, with the real-world execution on the left-hand side ("pegasus") and the simulated execution on the right-hand side ("wrench"). Tasks executions are shown on the vertical axis, each shown as a line segment along the horizontal time axis, spanning the time between the task's start time and the task's finish time. Different task types, i.e., different executables, are shown with different colors. In this workflow, all tasks of the same type are independent and have the same priority. We see that the shapes of the yellow regions, for example, vary between the two executions. These variations are explained by implementation-dependent behaviors of the workflow scheduler. In many instances throughout workflow execution several ready tasks can be selected for execution, e.g., sets of independent tasks in the same level of the workflow. When the number of available compute resources, n, is smaller than the number of ready tasks, the scheduler picks n ready tasks for immediate execution. In most WMSs, these tasks are picked as whatever first n tasks are returned when iterating over data structures in which task objects are stored. Building a perfectly faithful simulation of a WMS would thus entail implementing/using the exact same data structures as that in the actual implementation. This could be labor intensive or perhaps not even possible depending on which data structures, languages, and/or libraries are used in that implementation. In the context of this Pegasus case study, the production implementation of the DAGMan scheduler uses a custom priority list implementation to store ready tasks, while our simulation version of it

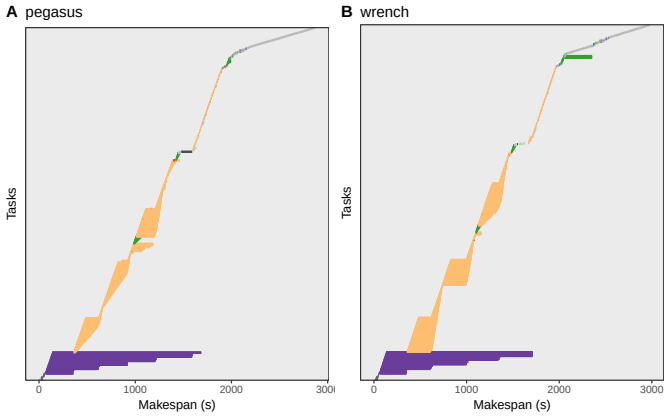

Figure 5: Task execution Gantt chart for sample real-world ("pegasus") and simulated ("wrench") executions of the Montage-2.0 workflow on the AWS-m5.xlarge platform.

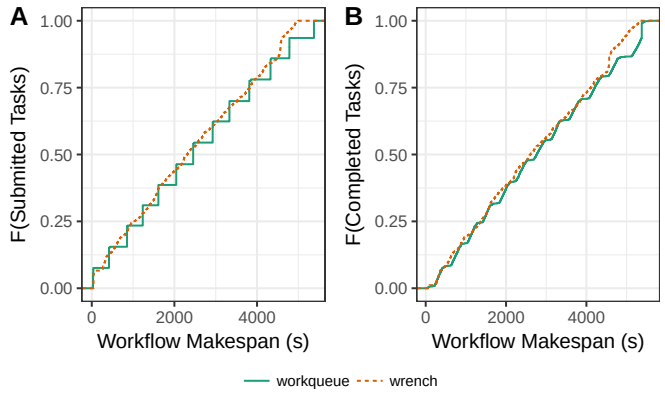

Figure 6: Empirical cumulative distribution function of task submit times (left) and task completion times (right) for sample real-world ("workqueue") and simulated ("wrench") executions of SAND on Chameleon Cloud.

stores workflow tasks in a C++ `std::map` data structure indexed by task string IDs. Consequently, when the real-world scheduler picks the first n ready tasks it typically picks different tasks than those picked by its simulated implementation. This is the cause the discrepancies seen in Figure 5.

4.5. WorkQueue: Simulation Accuracy

Similar to results obtained with our Pegasus simulator, we observe small (below 4%) average relative differences between actual and simulated makespans using WorkQueue for executing the SAND workflow on the Chameleon cloud platform (4th experimental scenario in Table 1). The two-sample KS tests for both task submissions and completions indicate that simulated workflow task executions statistically match real-world executions with the WorkQueue application execution framework. Figure 6 shows real-world ("workqueue") and simulated ("wrench") ECDFs of task submission, resp. completion, dates on the left-hand, resp. right-hand, side. The real-world and simulated task completion ECDFs closely match. However, for task submission dates, although the real-world and simulated ECDFs show very similar trends, the former exhibits a step function pattern while the latter does not. After investigating

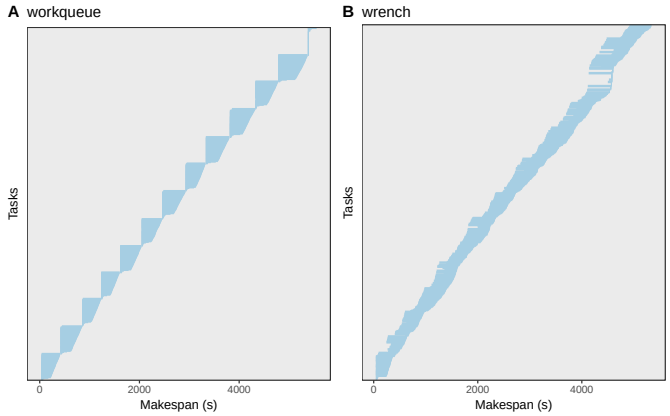

Figure 7: Task execution Gantt chart for sample real-world ("workqueue") and simulated ("wrench") executions of the SAND framework on the Chameleon Cloud platform.

this inconsistency, we found that in the real-world experiments execution events are recorded in logs as jobs are generated, but not when they are actually submitted to the HTCondor pool. As a result, our ground truth is biased for task submission dates as many tasks appear to be submitted at once. An additional (small) source of discrepancy is that in the real-world execution monitoring data from the HTCondor pool is pulled at regular intervals (about 5s), while in our simulated execution there is no such delay (this delay could be added by modifying our simulator's implementation). We hypothesize that the simulated execution may actually be closer to the real-world execution than what is reported in the real-world execution logs (verifying this hypothesis would require re-engineering the actual WorkQueue implementation to improve its logging feature). More generally, the above highlights the difficulties involved in defining what constitutes a sensible ground truth and obtaining this ground truth via real-world executions.

Figure 7 shows Gantt charts corresponding to the SAND executions shown in Figure 6. In the real-world execution, we observe the same step function pattern seen in Figure 6, for the same reasons. But we also note other discrepancies between real-world and simulated executions. For instance, at the end of the real-world execution, there is a gap in the Gantt chart. This gap corresponds to tasks with sub-second durations, which are not visible due to the chart's resolution. In the real-world executions, all these tasks are submitted in sequence, hence the gap. By contrast, in the simulated execution these tasks are not submitted in sequence. The reason is exactly the same phenomenon as that observed for Pegasus in the previous section. During workflow execution there are often more ready tasks than available workers, and WorkQueue must pick some of these tasks for submissions. This is done by removing the desired number of tasks from some data structure, and, here again the real-world WorkQueue implementation and our simulation of it use different such data structures: the former uses a `std::vector`, while the latter uses a `std::map`.

Overall, the result in this and in the previous sections show that WRENCH makes it possible to accurately simulate the ex-

ecution of scientific workflows on production platforms. This was demonstrated for two qualitatively different systems: the Pegasus WMS (which uses a push model to perform early-binding of tasks to compute resources) and the WorkQueue application execution framework (which uses a pull model to perform late-binding of tasks to compute resources).

4.6. Simulation Scalability

In this section, we evaluate the speed and the memory footprint of WRENCH simulations. Results with WRENCH version 1.2 were presented in the preliminary version of this work [48], while results presented in this section are obtained with version 1.4. Between these two versions a number of performance improvements were made, which explains why the results presented here are superior. About 30% of the performance gain is due to using more appropriate data structures (e.g., whenever possible replacing C++ `std::map` and `std::set` data structures by `std::unordered_map` and `std::unordered_set` since the latter have $O(1)$ average case operations). About 50% of the gain is due to the removal of a data structure to keep track of all in-flight (simulated) network messages, so as to avoid memory leaks when simulating host failures. This feature is now only activated whenever simulation of host failures is done. The remaining 20% is due to miscellaneous code improvements (e.g., avoiding pass-by-value parameters, using helper data structures to trade off space for time).

Table 2 shows average simulated makespans and simulation execution times for our 4 experimental scenarios. Simulations are executed on a single core of a MacBook Pro 3.5 GHz Intel Core i7 with 16GiB of RAM. For these scenarios, simulation times are more than 100x and up to 2500x shorter than real-world workflow executions. This is because SimGrid simulates computation and communication operations as delays computed based on computation and communication volumes using simulation models with low computational complexity.

To further evaluate the scalability of our simulator, we use a workflow generator [65] to generate representative randomized configurations of the Montage workflow with from $1,000$ up to $10,000$ tasks. We generate 5 workflow instances for each number of tasks, and simulate the execution of these generated workflow instances on 128 cores (AWS-m5.xlarge with 32 4-core nodes) using our WRENCH-based Pegasus simulator. Figure 8 shows simulation time (left vertical axis) and maximum resident set size (right vertical axis) vs. the number of tasks in the workflow. Each sample point is the average over the 5 workflow instances (error bars are shown as well). As expected, both simulation time and memory footprint increase as workflows become larger. The memory footprint grows linearly with the number of tasks (simply due to the need to store more task objects). The simulation time grows faster initially, but then linearly beyond 7,000 tasks. We conclude that the simulation scales well, making it possible to simulate very large 10,000-task Montage configurations in under 13 minutes on a standard laptop computer.

Figure 8 also includes results obtained with WorkflowSim. We find that WorkflowSim has a larger memory footprint than

Experimental Scenario			Avg. Workflow	Avg. Simulation
Workflow	System	Platform	Makespan (s)	Time (s)
1000Genome	Pegasus	ExoGENI	761.0 ±7.93	0.3 ±0.01
Montage-1.5	Pegasus	AWS-t2.xlarge	1,784.0 ±137.67	8.3 ±0.09
Montage-2.0	Pegasus	AWS-m5.xlarge	2,911.8 ±48.80	28.1 ±0.52
SAND	WorkQueue	Chameleon	5,339.2 ±133.56	16.3 ±0.86

Table 2: Simulated workflow makespans and simulation times averaged over 5 runs of each of our 4 experimental scenarios.

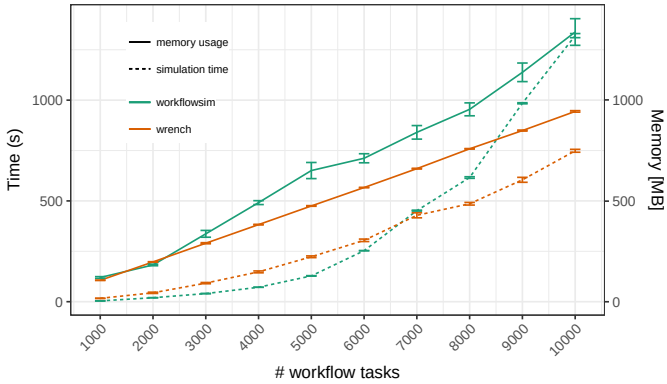

Figure 8: Average simulation time (in seconds. left vertical axis) and memory usage (maximum resident set size, right vertical axis) in MiB vs. workflow size.

our WRENCH-based simulator (by a factor ~1.41 for 10,000-task workflows), and it is slower than our WRENCH-based simulator (by a factor ~1.76 for 10,000-task workflows). In our previous work [48], WorkflowSim was faster than our WRENCH-based simulator (by a factor ~1.81 for 10,000-task workflows), with roughly similar trends. The reason for this improvement is a set of memory management optimizations that were applied to the WRENCH implementation, and in particular for handling message objects exchanged between processes. These optimizations have significantly improved the scalability of WRENCH, and it is likely for several other optimizations are possible to push scalability further. These optimizations have also decreased the workflow simulation time significantly when compared to the earlier WRENCH release, e.g., by a factor ~2.6 for 10,000-task workflows.

Overall, our experimental results show that WRENCH not only yields accurate simulation results but also can scalably simulate the execution of large-scale complex scientific applications running on heterogeneous platforms.

5. Conclusion

In this paper, we have presented WRENCH, a simulation framework for building simulators of Workflow Management Systems. WRENCH implements high-level simulation abstractions on top of the SimGrid simulation framework, so as to make it possible to build simulators that are accurate, that can run scalably on a single computer, and that can be implemented with minimal software development effort. Via case studies for the Pegasus production WMS and WorkQueue application execution framework, we have demonstrated that WRENCH achieves these objectives, and that it favorably compares to a

recently proposed workflow simulator. The main finding is that with WRENCH one can implement an accurate and scalable simulator of a complex real-world system with a few hundred lines of code. WRENCH is open source and welcomes contributors. WRENCH is already being used for several research and education projects, and Version 1.5 was released in February 2020. We refer the reader to `https://wrench-project.org` for software, documentation, and links to related projects.

A short-term development direction is to use WRENCH to simulate the execution of current production WMSs and application execution frameworks (as was done for Pegasus and WorkQueue in Section 4). Although we have designed WRENCH with knowledge of many such systems in mind, we expect that WRENCH APIs and abstractions will evolve once we set out to realize these implementations. Another development direction is the implementation of more CI service abstractions in WRENCH, e.g., a Hadoop Compute Service, specific distributed cloud Storage Services. From a research perspective, a future direction is that of automated simulation calibration. As seen in our Pegasus and WorkQueue case studies, even when using validated simulation models, the values of a number of simulation parameters must be carefully chosen in order to obtain accurate simulation results. This issue is not confined to WRENCH, but is faced by all distributed system simulators. In our case studies, we have calibrated these parameters manually by analyzing and comparing simulated and real-world execution event traces. While, to the best of our knowledge, this is the typical practice, what is truly needed is an automated calibration method. Ideally, this method would process a (small) number of (not too large) real-world execution traces for "training scenarios", and compute a valid and robust set of calibration parameter values. An important research question will then be to understand to which extent these automatically computed calibrations can be composed and extrapolated to scenarios beyond the training scenarios.

Acknowledgments

This work is funded by NSF contracts #1642369 and #1642335, "SI2-SSE: WRENCH: A Simulation Workbench for Scientific Worflow Users, Developers, and Researchers"; by CNRS under grant #PICS07239; and partly funded by NSF contracts #1923539 and #1923621: "CyberTraining: Implementation: Small: Integrating core CI literacy and skills into university curricula via simulation-driven activities". We thank Martin Quinson, Arnaud Legrand, and Pierre-François Dutot for their valuable help. We also thank the NSF Chameleon Cloud for providing time grants to access their resources.

References

References

[1] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields, Workflows for e-Science: scientific workflows for grids, Springer Publishing Company, Incorporated, 2007. doi:10.1007/978-1-84628-757-2.

[2] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus: a Workflow Management System for Science Automation, Future Generation Computer Systems 46 (2015) 17–35. doi:10.1016/j.future.2014.10.008.

[3] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, et al., Askalon: A development and grid computing environment for scientific workflows, in: Workflows for e-Science, Springer, 2007, pp. 450–471. doi:10.1007/978-1-84628-757-2_27.

[4] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, I. Foster, Swift: A language for distributed parallel scripting, Parallel Computing 37 (9) (2011) 633–652. doi:10.1016/j.parco.2011.05.005.

[5] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al., The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud, Nucleic acids research (2013) gkt328doi:10.1093/nar/gkt328.

[6] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock, Kepler: an extensible system for design and execution of scientific workflows, in: Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on, IEEE, 2004, pp. 423–424. doi:10.1109/SSDM.2004.1311241.

[7] M. Albrecht, P. Donnelly, P. Bui, D. Thain, Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids, in: 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, ACM, 2012, p. 1. doi:10.1145/2443416.2443417.

[8] N. Vydyanathan, U. V. Catalyurek, T. M. Kurc, P. Sadayappan, J. H. Saltz, Toward optimizing latency under throughput constraints for application workflows on clusters, in: Euro-Par 2007 Parallel Processing, Springer, 2007, pp. 173–183. doi:10.1007/978-3-540-74466-5_20.

[9] A. Benoit, V. Rehn-Sonigo, Y. Robert, Optimizing latency and reliability of pipeline workflow applications, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE, 2008, pp. 1–10. doi:10.1109/IPDPS.2008.4536160.

[10] Y. Gu, Q. Wu, Maximizing workflow throughput for streaming applications in distributed environments, in: Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on, IEEE, 2010, pp. 1–6. doi:10.1109/ICCCN.2010.5560146.

[11] M. Malawski, G. Juve, W. Deelman, J. Nabrzyski, Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds, Future Generation Computer Systems 48 (2015) 1–18. doi:10.1016/j.future.2015.01.004.

[12] J. Chen, Y. Yang, Temporal dependency-based checkpoint selection for dynamic verification of temporal constraints in scientific workflow systems, ACM Transactions on Software Engineering and Methodology (TOSEM) 20 (3) (2011) 9. doi:10.1145/2000791.2000793.

[13] G. Kandaswamy, A. Mandal, D. Reed, et al., Fault tolerance and recovery of scientific workflows on computational grids, in: Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on, IEEE, 2008, pp. 777–782. doi:10.1109/CCGRID.2008.79.

[14] R. Ferreira da Silva, T. Glatard, F. Desprez, Self-healing of workflow activity incidents on distributed computing infrastructures, Future Generation Computer Systems 29 (8) (2013) 2284–2294. doi:10.1016/j.future.2013.06.012.

[15] W. Chen, R. Ferreira da Silva, E. Deelman, T. Fahringer, Dynamic and fault-tolerant clustering for scientific workflows, IEEE Transactions on Cloud Computing 4 (1) (2016) 49–62. doi:10.1109/TCC.2015.2427200.

[16] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, T. Fahringer, A multi-objective approach for workflow scheduling in heterogeneous environments, in: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), IEEE Computer Society, 2012, pp. 300–309. doi:10.1109/CCGrid.2012.114.

[17] I. Pietri, M. Malawski, G. Juve, E. Deelman, J. Nabrzyski, R. Sakellariou, Energy-constrained provisioning for scientific workflow ensembles, in: Cloud and Green Computing (CGC), 2013 Third International Conference on, IEEE, 2013, pp. 34–41. doi:10.1109/CGC.2013.14.

[18] M. Tikir, M. Laurenzano, L. Carrington, A. Snavely, PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications, in: Proc. of the 15th Intl. Euro-Par Conf. on Parallel Processing,

no. 5704 in LNCS, Springer, 2009, pp. 135–148. doi:10.1007/978-3-642-03869-3_16.

[19] T. Hoefler, T. Schneider, A. Lumsdaine, LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model, in: Proc. of the ACM Workshop on Large-Scale System and Application Performance, 2010, pp. 597–604. doi:10.1145/1851476.1851564.

[20] G. Zheng, G. Kakulapati, L. Kalé, BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines, in: Proc. of the 18th Intl. Parallel and Distributed Processing Symposium (IPDPS), 2004. doi:10.1109/IPDPS.2004.1303013.

[21] R. Bagrodia, E. Deelman, T. Phan, Parallel Simulation of Large-Scale Parallel Applications, International Journal of High Performance Computing Applications 15 (1) (2001) 3–12. doi:10.1177/109434200101500101.

[22] W. H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, F. Zini, OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies, International Journal of High Performance Computing Applications 17 (4) (2003) 403–416. doi:10.1177/10943420030174005.

[23] R. Buyya, M. Murshed, GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, Concurrency and Computation: Practice and Experience 14 (13-15) (2003) 1175–1220. doi:10.1002/cpe.710.

[24] S. Ostermann, R. Prodan, T. Fahringer, Dynamic Cloud Provisioning for Scientific Grid Workflows, in: Proc. of the 11th ACM/IEEE Intl. Conf. on Grid Computing (Grid), 2010, pp. 97–104. doi:10.1109/GRID.2010.5697953.

[25] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms, Software: Practice and Experience 41 (1) (2011) 23–50. doi:10.1002/spe.995.

[26] A. Núñez, J. Vázquez-Poletti, A. Caminero, J. Carretero, I. M. Llorente, Design of a New Cloud Computing Simulation Platform, in: Proc. of the 11th Intl. Conf. on Computational Science and its Applications, 2011, pp. 582–593. doi:10.1007/978-3-642-21931-3_45.

[27] G. Kecskemeti, DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds, Simulation Modelling Practice and Theory 58 (2) (2015) 188–218. doi:10.1016/j.simpat.2015.05.009.

[28] A. Montresor, M. Jelasity, PeerSim: A Scalable P2P Simulator, in: Proc. of the 9th Intl. Conf. on Peer-to-Peer, 2009, pp. 99–100. doi:10.1109/P2P.2009.5284506.

[29] I. Baumgart, B. Heep, S. Krause, OverSim: A Flexible Overlay Network Simulation Framework, in: Proc. of the 10th IEEE Global Internet Symposium, IEEE, 2007, pp. 79–84. doi:10.1109/GI.2007.4301435.

[30] M. Taufer, A. Kerstens, T. Estrada, D. Flores, P. J. Teller, SimBA: A Discrete Event Simulator for Performance Prediction of Volunteer Computing Projects, in: Proc. of the 21st Intl. Workshop on Principles of Advanced and Distributed Simulation, 2007, pp. 189–197. doi:10.1109/PADS.2007.27.

[31] T. Estrada, M. Taufer, K. Reed, D. P. Anderson, EmBOINC: An Emulator for Performance Analysis of BOINC Projects, in: Proc. of the Workshop on Large-Scale and Volatile Desktop Grids (PCGrid), 2009. doi:10.1109/IPDPS.2009.5161135.

[32] D. Kondo, SimBOINC: A Simulator for Desktop Grids and Volunteer Computing Systems, Available at http://simboinc.gforge.inria.fr/ (2007).

[33] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms, Journal of Parallel and Distributed Computing 74 (10) (2014) 2899–2917. doi:10.1016/j.jpdc.2014.06.008.

[34] C. D. Carothers, D. Bauer, S. Pearce, ROSS: A High-Performance, Low Memory, Modular Time Warp System, in: Proc. of the 14th ACM/IEEE/SCS Workshop of Parallel on Distributed Simulation, 2000, pp. 53–60. doi:10.1109/PADS.2000.847144.

[35] W. Chen, E. Deelman, WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments, in: Proc. of the 8th IEEE Intl. Conf. on E-Science, 2012, pp. 1–8. doi:10.1109/eScience.2012.6404430.

[36] A. Hirales-Carbajal, A. Tchernykh, T. Röblitz, R. Yahyapour, A Grid simulation framework to study advance scheduling strategies for complex workflow applications, in: In Proc. of IEEE Intl. Symp. on Parallel Distributed Processing Workshops (IPDPSW), 2010. doi:10.1109/IPDPSW.2010.5470918.

[37] M.-H. Tsai, K.-C. Lai, H.-Y. Chang, K. Fu Chen, K.-C. Huang, Pewss: A platform of extensible workflow simulation service for workflow scheduling research, Software: Practice and Experience 48 (4) (2017) 796–819. doi:10.1002/spe.2555.

[38] S. Ostermann, K. Plankensteiner, D. Bodner, G. Kraler, R. Prodan, Integration of an Event-Based Simulation Framework into a Scientific Workflow Execution Environment for Grids and Clouds, in: In proc. of the 4th ServiceWave European Conference, 2011, pp. 1–13. doi:10.1007/978-3-642-24755-2_1.

[39] G. Kecskemeti, S. Ostermann, R. Prodan, Fostering Energy-Awareness in Simulations Behind Scientific Workflow Management Systems, in: Proc. of the 7th IEEE/ACM Intl. Conf. on Utility and Cloud Computing, 2014, pp. 29–38. doi:10.1109/UCC.2014.11.

[40] J. Cao, S. Jarvis, S. Saini, G. Nudd, GridFlow: Workflow Management for Grid Computing, in: Proc. of the 3rd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGrid), 2003, pp. 198–205.

[41] The SimGrid Project, Available at http://simgrid.org/ (2019).

[42] P. Bedaride, A. Degomme, S. Genaud, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, F. Suter, B. Videau, Toward Better Simulation of MPI Applications on Ethernet/TCP Networks, in: Prod. of the 4th Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2013. doi:10.1007/978-3-319-10214-6_8.

[43] P. Velho, L. Mello Schnorr, H. Casanova, A. Legrand, On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations, ACM Transactions on Modeling and Computer Simulation 23 (4) (2013). doi:10.1145/2517448.

[44] P. Velho, A. Legrand, Accuracy Study and Improvement of Network Simulation in the SimGrid Framework, in: Proc. of the 2nd Intl. Conf. on Simulation Tools and Techniques, 2009. doi:10.4108/ICST.SIMUTOOLS2009.5592.

[45] K. Fujiwara, H. Casanova, Speed and Accuracy of Network Simulation in the SimGrid Framework, in: Proc. of the 1st Intl. Workshop on Network Simulation Tools, 2007.

[46] A. Lèbre, A. Legrand, F. Suter, P. Veyre, Adding Storage Simulation Capacities to the SimGrid Toolkit: Concepts, Models, and API, in: Proc. of the 8th IEEE Intl. Symp. on Cluster Computing and the Grid, 2015. doi:10.1109/CCGrid.2015.134.

[47] The WRENCH Project, https://wrench-project.org (2020).

[48] H. Casanova, S. Pandey, J. Oeth, R. Tanaka, F. Suter, R. Ferreira da Silva, WRENCH: A Framework for Simulating Workflow Management Systems, in: 13th Workshop on Workflows in Support of Large-Scale Science (WORKS'18), 2018, pp. 74–85. doi:10.1109/WORKS.2018.00013.

[49] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, D. Thain, Harnessing parallelism in multicore clusters with the all-pairs, wavefront, and makeflow abstractions, Cluster Computing 13 (3) (2010) 243–256. doi:10.1007/s10586-010-0134-7.

[50] The ns-3 Network Simulator, Available at http://www.nsnam.org.

[51] E. León, R. Riesen, A. Maccabe, P. Bridges, Instruction-Level Simulation of a Cluster at Scale, in: Proc. of the Intl. Conf. for High Performance Computing and Communications (SC), 2009. doi:10.1145/1654059.1654063.

[52] R. Fujimoto, Parallel Discrete Event Simulation, Commun. ACM 33 (10) (1990) 30–53. doi:10.1145/84537.84545.

[53] V. Cima, J. Beránek, S. Böhm, ESTEE: A Simulation Toolkit for Distributed Workflow Execution, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2019, research Poster.

[54] S. Ostermann, G. Kecskemeti, R. Prodan, Multi-layered Simulations at the Heart of Workflow Enactment on Clouds, Concurrency and Computation Practice and Experience 28 (2016) 3180—3201. doi:10.1002/cpe.3733.

[55] R. Matha, S. Ristov, R. Prodan, Simulation of a workflow execution as a real Cloud by adding noise, Simulation Modelling Practice and Theory 79 (2017) 37–53. doi:10.1016/j.simpat.2017.09.003.

[56] L. Bobelin, A. Legrand, D. A. G. Márquez, P. Navarro, M. Quinson, F. Suter, C. Thiery, Scalable Multi-Purpose Network Representation for

Large Scale Distributed System Simulation, in: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Ottawa, Canada, 2012, pp. 220–227. doi:10.1109/CCGrid.2012.31.

[57] M. Turilli, M. Santcroos, S. Jha, A Comprehensive Perspective on Pilot-Job Systems, ACM Comput. Surv. 51 (2) (2018) 43:1–43:32. doi:10.1145/3177851.

[58] J. Frey, Condor dagman: Handling inter-job dependencies, Tech. rep. (2002).
URL http://www.bo.infn.it/calcolo/condor/dagman/

[59] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience, Concurrency and computation: practice and experience 17 (2-4) (2005) 323–356. doi:10.1002/cpe.938.

[60] B. Tovar, R. Ferreira da Silva, G. Juve, E. Deelman, W. Allcock, D. Thain, M. Livny, A job sizing strategy for high-throughput scientific workflows, IEEE Transactions on Parallel and Distributed Systems 29 (2) (2018) 240–253. doi:10.1109/TPDS.2017.2762310.

[61] The WRENCH Pegasus Simulator, https://github.com/wrench-project/pegasus (2019).

[62] The WRENCH WorkQueue Simulator, https://github.com/wrench-project/workqueue (2019).

[63] R. Ferreira da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira, I. M. Overton, M. Atkinson, Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows, Future Generation Computer Systems 95 (2019) 615–628. doi:10.1016/j.future.2019.01.015.

[64] C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, D. Thain, A framework for scalable genome assembly on clusters, clouds, and grids, IEEE Transactions on Parallel and Distributed Systems 23 (12) (2012) 2189–2197. doi:10.1109/TPDS.2012.80.

[65] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, E. Deelman, Community resources for enabling and evaluating research on scientific workflows, in: 10th IEEE International Conference on e-Science, eScience'14, 2014, pp. 177–184. doi:10.1109/eScience.2014.44.

[66] Pegasus' DAX Workflow Description Format, https://pegasus.isi.edu/documentation/creating_workflows.php (2019).

[67] The Standard Workload Format, http://www.cs.huji.ac.il/labs/parallel/workload/swf.html (2019).

[68] D. Lifka, The ANL/IBM SP Scheduling System, in: Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing, LCNS, Vol. 949, 1995, pp. 295–303. doi:10.1007/3-540-60153-8_35.

[69] F. Dabek, R. Cox, F. Kaashoek, R. Morris, Vivaldi: A Decentralized Network Coordinate System, in: Proc. of SIGCOMM, 2004. doi:10.1145/1015467.1015471.

Appendix A. Example WRENCH Simulator

An example WRENCH simulator developed using the WRENCH User API (see Section 3.5) is shown in Figure 9. This simulator uses a WMS implementation (called `SomeWMS`) that has already been developed using the WRENCH Developer API (see Section 3.4). After initializing the simulation (lines 7-8), the simulator instantiates a platform (line 11) and a workflow (line 14-15). A workflow is defined as a set of computation tasks and data files, with control and data dependencies between tasks. Each task can also have a priority, which can then be taken into account by a WMS for scheduling purposes. Although the workflow can be defined purely programmatically, in this example the workflow is imported from a workflow description file in the DAX format [66]. At line 18 the simulator creates a storage service with 1PiB capacity accessible on host `storage_host`. This and other hostnames are specified in the XML platform description file. At line 22 the simulator creates a compute service that corresponds to a 4-node batch-scheduled cluster. The physical characteristics of the compute nodes (`node[1-4]`) are specified in the platform description file. This compute service has a 1TiB scratch storage space. Its behavior is customized by passing a couple of property-value pairs to its constructor. It will be subject to a background load as defined by a trace in the standard SWF format [67], and its batch queue will be managed using the EASY Backfilling scheduling algorithm [68]. The simulator then creates a second compute service (line 28), which is a 4-host cloud service with 4TiB scratch space, customized so that it does not support pilot jobs. Two helper services are instantiated, a data registry service so that the WMS can keep track of file locations (line 33) and a network monitoring service that uses the Vivaldi algorithm [69] to measure network distances between the two hosts from which the compute services are accessed (`batch_login` and `cloud_gateway`) and the `my_host` host, which is the host that runs these helper services and the WMS (line 36). At line 41, the simulator specifies that the workflow data file `input_file` is initially available at the storage service. It then instantiates the WMS and passes to it all available services (line 44), and assigns the workflow to it (line 47). The crucial call is at line 50, where the simulation is launched and the simulator hands off control to WRENCH. When this call returns the workflow has either completed or failed. Assuming it has completed, the simulator then retrieves the ordered set of task completion events (line 53) and performs some (in this example, trivial) mining of these events (line 55).

For brevity, the example in Figure 9 omits `try/catch` clauses. Also, note that although the simulator uses the `new` operator to instantiate WRENCH objects, the simulation object takes ownership of these objects (using unique or shared pointers), so that there is no memory deallocation onus placed on the user.

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

int main(int argc, char **argv) {

  // Declare and initialize a simulation
  wrench::Simulation simulation;
  simulation.init(&argc, argv);

  // Instantiate a platform
  simulation.instantiatePlatform("my_platform.xml");

  // Instantiate a workflow
  wrench::Workflow workflow;
  workflow.loadFromDAX("my_workflow.dax", "1000Gf");

  // Instantiate a storage service
  auto storage_service = simulation.add(new wrench::SimpleStorageService("storage_host", pow(2,50)));

  // Instantiate a sompute service (a batch-scheduled 4-node cluster that uses the
  // EASY backfilling algorithm and is subject to a background load)
  auto batch_service = simulation.add(
      new wrench::BatchService("batch_login", {"node1", "node2", "node3", "node4"}, pow(2,40),
          {{wrench::BatchServiceProperty::SIMULATED_WORKLOAD_TRACE_FILE, "load.swf"},
           {wrench::BatchServiceProperty::BATCH_SCHEDULING_ALGORITHM, "easy_bf"}}));

  // Instantiate a compute service (a 4-host cloud platform that does not support pilot jobs)
  auto cloud_service = simulation.add(
      new wrench::CloudService("cloud_gateway", {"host1", "host2", "host3", "host4"}, pow(2,42),
          {{wrench::CloudServiceProperty::SUPPORTS_PILOT_JOBS, "false"}}));

  // Instantiate a data registry service
  auto data_registry_service = simulation.add(new wrench::FileRegistryService("my_desktop"));

  // Instantiate a network monitoring service
  auto network_monitoring_service = simulation.add(new wrench::NetworkProximityService(
      "my_desktop", {"my_desktop", "batch_login", "cloud_gateway"},
      {{wrench::NetworkProximityServiceProperty::NETWORK_PROXIMITY_SERVICE_TYPE, "vivaldi"}});

  // Stage a workflow input file at the storage service
  simulation.stageFile(workflow.getFileByID("input_file"), storage_service);

  // Instantiate a WMS...
  auto wms = simulation.add(new wrench::SomeWMS({batch_service, cloud_service}, {storage_service},
      {network_monitoring_service}, {data_registry_service}, "my_desktop"));
  // ... and assign the workflow to it, to be executed one hour in
  wms->addWorkflow(&workflow, 3600);

  // Launch the simulation
  simulation.launch();

  // Retrieve task completion events
  auto trace = simulation.getOutput().getTrace<wrench::SimulationTimestampTaskCompletion>();
  // Determine the completion time of the last task that completed
  double completion_time = trace[trace.size()-1]->getContent()->getDate();
}
```

Figure 9: Example fully functional WRENCH simulator. Try-catch clauses are omitted.