SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications

Gabriel A. Moreno
gmoreno@sei.cmu.edu
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, PA, USA

Bradley Schmerl
schmerl@cs.cmu.edu
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA, USA

David Garlan
garlan@cs.cmu.edu
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA, USA

ABSTRACT

Research in self-adaptive systems often uses web applications as target systems, running the actual software on real web servers. This approach has three drawbacks. First, these systems are not easy and/or cheap to deploy. Second, run-time conditions cannot be replicated exactly to compare different adaptation approaches due to uncontrolled factors. Third, running experiments is time-consuming. To address these issues, we present SWIM, an exemplar that simulates a web application. SWIM can be used as a target system with an external adaptation manager interacting with it through its TCP-based interface. Since the servers are simulated, this use case addresses the first two problems. The full benefit of SWIM is attained when the adaptation manager is built as a simulation module. An experiment using a simulated 60-server cluster, processing 18 hours of traffic with 29 million requests takes only 5 minutes to run on a laptop computer. SWIM has been used for evaluating self-adaptation approaches, and for a comparative study of model-based predictive approaches to self-adaptation.

ACM Reference Format:
Gabriel A. Moreno, Bradley Schmerl, and David Garlan. 2018. SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194133.3194163

1 INTRODUCTION

Research in self-adaptive systems often uses web applications like Znn.com [5] and RUBiS [25] as target systems [4, 7, 9, 12, 13, 15–19, 24, 26, 28]. In most cases, these systems are configured to use multiple web servers so that the self-adaptation manager can increase or decrease the number of active servers to deal with changing traffic to the application. Although these applications were developed for research and benchmarking, they are not simulated—they run actual code and process requests on real web servers such as the Apache HTTP Server. This complicates the deployment of these experimental systems because it requires having a distributed system of physical or virtual machines to deploy the web servers. And even if virtual machines are commissioned from a cloud provider to simplify the deployment, there is still a cost to run the experiments.

Another problem with running the software on real web servers is that the run-time conditions cannot be replicated exactly to compare different self-adaptation approaches due to several uncontrollable factors, including the processing speed of the servers, background processes, and network delays. This prevents comparing results obtained by different research groups, and even between experiments run by the same researcher.

Yet another problem is that running experiments is time consuming. These experiments involve sending requests to the system for some period of time long enough to allow changes in the traffic to the system. Ideally, the traffic to the system is generated from a prerecorded trace captured from real traffic to a web application. In such cases, the traces may have to be several hours long in order to capture naturally-occurring changes in the traffic. Depending on the experiment design, running a complete set of experiments may take days. Furthermore, even while self-adaptation approaches are being developed and tested, each run may be inconveniently long.

To address these issues, we present the Simulator for Web Infrastructure and Management (SWIM), an exemplar that simulates a web application like Znn.com or RUBiS. SWIM simulates the processing of requests in multiple servers, but runs as a single process in a single computer, making the deployment for an experiment simple regardless of the number of servers required. Although there are several random elements in the simulation, such as the time to process a request, the random number generators are seeded, thus allowing the replication of the same conditions multiple times.

The interaction between the adaptation manager and SWIM can be done in two different ways. In the first mode, SWIM can be used as a target system with an external adaptation manager interacting with it through its TCP-based interface. This makes the integration with SWIM straightforward regardless of the language in which the adaptation manager was developed. With this use case using an external adaptation manager, SWIM addresses the first two problems described above. The full benefit of SWIM is attained when the adaptation manager is built as a simulation module, because in this case SWIM can control the timing of the interaction with the adaptation manager, and thus it can compress time, by skipping the wait time between events. In this second mode, an experiment using a simulated 60-server cluster, processing 18 hours of traffic with 29 million requests takes only 5 minutes to run on a laptop computer.

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194163

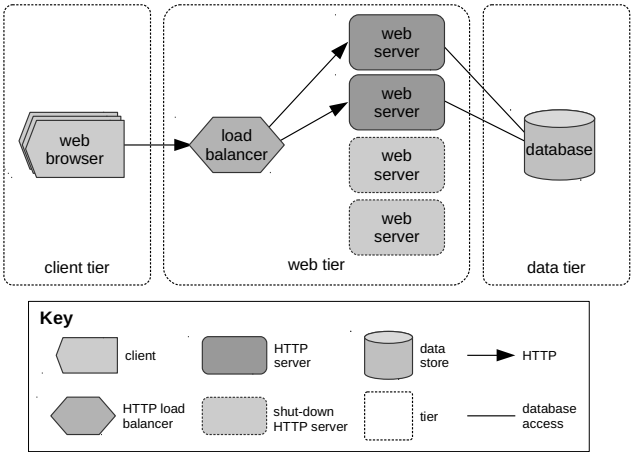

Figure 1: Simulated web application architecture.

SWIM has been used for evaluating self-adaptation approaches [21], and for a comparative study of model-based predictive approaches to self-adaptation [20]. Cito and Gall propose using Docker containers for artifacts supporting software engineering research [6], so we are now making this exemplar available as a Docker image for easy deployment on different operating systems.[1] This distribution includes: two simple adaptation managers that show how other adaptation managers can be developed for SWIM; the traces used for the comparative study; and a tool to create plots of the simulation results.

The rest of the paper is organized as follows. Section 2 provides and overview of SWIM and the kind of web applications it simulates. Section 3 describes how adaptation managers for SWIM are implemented. A description of how experiments with SWIM are run is presented in Section 4.

2 OVERVIEW OF SWIM

SWIM simulates a generic multi-tier web application similar to Znn.com and RUBiS. An application of this kind consists of a web server tier that receives requests from clients using browsers, and a database tier. A load balancer is used to support multiple servers in the web tier, as shown in Figure 1. The load balancer distributes the requests arriving at the website among the web servers. When a client requests a web page using a browser, the web server processing the request accesses the database tier to get the data needed to render the page with dynamic content. The request arrival rate, which induces the workload on the system, changes over time, and we want the system to be able to self-adapt to best deal with this changing environment.

There are two ways the system can deal with changes in the workload induced by the clients. First, as is typical in elastic computing, the system can add/remove servers to/from the pool of servers connected to the load balancer. Second, it can control whether responses include optional content (e.g., advertisement or suggested products) or not, since not including the optional content in the response to a request reduces the load imposed on the system. To this end, SWIM has *brownout* capability [16]. Instead of being limited to a binary choice in which all or no responses include the optional content, the brownout paradigm uses a control known as *dimmer* to control the proportion of responses that include the optional content, with 1 being the setting in which all responses include the optional content, 0 when no one does (i.e., blackout). The value of the dimmer can be thought of as the probability of a response including the optional content, thus taking values in [0..1].

In general, the cost of running a system is proportional to the number of servers used, and the revenue is higher when more optional content is served and the response time requirement is met. This introduces a trade-off that the adaptation manager has to manage. SWIM provides the necessary monitoring data to include these factors in the adaptation decision, and to compare adaptation approaches using utility functions that use these factors.

SWIM was implemented using OMNeT++, an extensible discrete event simulation environment [27]. SWIM does not simulate the network, the actual functionality of the web application, or what particular pages a user accesses. Instead, it only simulates the processing of requests at a high level—simply as a computation that takes time to execute—which is sufficient to evaluate approaches in terms of qualities such as response times, response types, and number of servers used.

The traffic to the system is simulated by replaying request traces stored in files. A trace file consists of one line per request, and each line is a single floating-point number that specifies how long to wait (in seconds) before sending the next request. SWIM includes traces from the WorldCup '98 trace archive [2] and the ClarkNet traces [3], which were previously recorded from real websites.

The requests arrive at the load balancer, and are forwarded to one of the servers following a round robin algorithm. Each server simulates the processing of requests in the web server. Akin to the maximum number of processes in a real web server, the maximum number of concurrent requests in a server is configurable in the simulation. When more than one request is being processed by a server, the sharing of its processor is simulated by inflating its processing time accordingly. Requests assigned to a server that is already processing the maximum number of concurrent request are queued and serviced in FIFO order.

The processing of a request is simulated only in terms of the time that it takes, not the computation or results it produces. This time is determined by the service time—the amount of time processing the request would take if there were no contention. To support brownout, when a request arrives at a server, a random number is drawn from a uniform distribution to determine whether its response should include the optional content according to the current dimmer setting. The service time is then drawn from a normal distribution whose parameters depend on the type of response. Although the parameters of the two distributions are configurable in a flexible way, generally the service time for responses that include the optional content have higher mean and variance.

In SWIM, the effect of a cold cache is also simulated by increasing service times when a server is newly instantiated, emulating how cache misses add to the normal steady state service time. As the server processes more requests, this effect gradually disappears.

All the random number generators used in the simulation are seeded so that it is possible to replicate experiments with the same

[1] Available at https://hub.docker.com/r/gabrielmoreno/swim/.

conditions, and thus make a direct comparison of different adaptation approaches.

Thus far, we have described SWIM as a target system. The next section describes how adaptation managers interact with SWIM.

3 IMPLEMENTING ADAPTATION MANAGERS

An adaptation manager requires an interface with the target system in order to monitor that system and to execute adaptation actions on it. SWIM provides two interfaces that allow an adaptation manager to interact with it. One is TCP-based and can be used by an external adaptation manager; the other requires that the adaptation manager be built as an OMNeT++ module. Both interfaces provide probing methods to obtain information about the system and its environment, including current dimmer value, number of servers and active servers,[2] utilization of each active server, average request arrival rate, and average throughput and response time for the two kinds of responses (i.e., with and without optional content). In addition, these interfaces have effectors that allow the adaptation manager to change the dimmer setting, and to add[3] and remove servers. All operations have negligible execution time, except for adding a server, which takes an amount of time configurable in the simulation. This time simulates the time it takes to boot a server, or instantiate a new virtual machine in the cloud.

In the following sections, the two alternatives for implementing adaptation managers using these interfaces are described.

3.1 Adaptation Manager as Simulation Module

To obtain the most benefit out of SWIM, the adaptation manager must be built as an OMNeT++ simulation module. Doing so allows the simulation to control the timing of events without needing to synchronize with an external adaptation manager process. This allows time to be compressed in two ways in the simulation. First, there is never a real wait for something to happen, since the simulation clock can be advanced to the next relevant event. In addition, operations that take time and whose actual result is not needed can be simulated by calculating when the operation would be completed and just inserting a completion event at that time into the future, without actually performing the time-consuming operation.

When used in this way, SWIM implements most of the MAPE-K loop [14]. The *model* module holds the knowledge of the system, including static information about configuration of the simulation (e.g., maximum number of servers supported), and information that changes during the execution of the system, which is collected by the *monitor* module. This module has probes to measure request arrival rate, throughput, server utilization, etc. The *execution manager* takes an adaptation tactic (i.e., add server, remove server, set dimmer, or a composite of them) and executes it to effect changes on the system. Analysis and planning are the only activities of the MAPE-K loop that an adaptation manager needs to provide as an OMNeT++ module.

An OMNeT++ module consists of a C++ class and a NED file that defines the simulation module type.[4] SWIM provides a class

```
1  Tactic* ReactiveAdaptationManager::evaluate() {
2      MacroTactic* pMacroTactic = new MacroTactic;
3      Model* pModel = getModel();
4      const double dimmerStep = 1.0 / (pModel->getNumberOfDimmerLevels() - 1);
5      double dimmerFactor = pModel->getDimmerFactor();
6      double spareUtilization = pModel->getConfiguration().getActiveServers() -
               pModel->getObservations().utilization;
7      bool isServerBooting = pModel->getServers() > pModel->getActiveServers();
8      double responseTime = pModel->getObservations().avgResponseTime;
9
10     if (responseTime > RT_THRESHOLD) {
11         if (!isServerBooting
12                 && pModel->getServers() < pModel->getMaxServers()) {
13             pMacroTactic->addTactic(new AddServerTactic);
14         } else if (dimmerFactor > 0.0) {
15             dimmerFactor = max(0.0, dimmerFactor - dimmerStep);
16             pMacroTactic->addTactic(new SetDimmerTactic(dimmerFactor));
17         }
18     } else if (responseTime < RT_THRESHOLD) { // can we increase dimmer or remove servers?
19
20         // only if there is more than one server of spare capacity
21         if (spareUtilization > 1) {
22             if (dimmerFactor < 1.0) {
23                 dimmerFactor = min(1.0, dimmerFactor + dimmerStep);
24                 pMacroTactic->addTactic(new SetDimmerTactic(dimmerFactor));
25             } else if (!isServerBooting
26                     && pModel->getServers() > 1) {
27                 pMacroTactic->addTactic(new RemoveServerTactic);
28             }
29         }
30     }
31     return pMacroTactic;
32 }
```

Listing 1: Adaptation manager as simulation module.

`BaseAdaptationManager` that handles the interaction with the other modules in the simulation. An adaptation manager simply needs to extend this class and implement the method `evaluate()`. The self-adaptation loop has a period defined with the configuration parameter `evaluationPeriod`.[5] In each period, the *monitor* module updates the *model*, and then invokes the `evaluate()` method of the adaptation manager. This method returns a pointer to an instance of `Tactic`, which is then passed to the *execution manager*. SWIM does not require separating the implementations of the evaluation and planning activities, but that can be done if desired.

Listing 1 shows the implementation of a simple reactive adaptation manager that tries to maintain the average response time below a threshold using the least number of servers, and serving the most optional content. The logic of this adaptation manager is as follows. If the observed average response time is greater than the threshold, it adds a server if possible, otherwise, it decreases the dimmer. If the response time is below the threshold and there is more than one server of active spare capacity, it increases the dimmer, but if the dimmer is already at its maximum, it removes a server if possible. The `evaluate()` method in this case returns a `MacroTactic`, which implements a composite pattern for tactics [8]. Although not used in this example, all the tactics contained by this tactic are executed in parallel, so that it is possible to add a server and change the dimmer setting simultaneously, for example. A `MacroTactic` can be empty if no adaptation is needed. Line 3 shows how the code gets access to the *model* to get all the information needed to make the adaptation decision.

Although OMNeT++ modules are written in C++, it is possible to use wrappers to invoke adaptation managers developed in

[2] A server is considered *active* if it has finished booting and can process requests.
[3] One current limitation of SWIM is that only one server can be booting at a time, and no server can be removed while another one is booting.
[4] Due to space constraints, it is not possible to cover details of OMNeT++ here. We refer the interested reader to the OMNeT++ manual [22].
[5] Configuration parameters are defined in an `.ini` file (see Section 4.1).

other languages as long as they can be invoked synchronously from `evaluate()`. Although the execution of the simulation stops while the adaptation manager is performing the evaluation, SWIM measures the time `evaluate()` takes to execute, and then simulates events as if the evaluation would have been performed in parallel with the system running. This is important for evaluating the effect decision delays have on the effectiveness of self-adaptation.

3.2 Adaptation Manager as External Program

SWIM's TCP-based interface can be used for adaptation managers that run as an external program. Although this has the drawback of having to run SWIM in wall-clock time, it has the advantage of allowing SWIM to be used as a target system for a variety of adaptation managers, regardless of how they are implemented, since most languages support communication through TCP sockets. SWIM has been used in this mode with Rainbow [10] and CobRA [1], which were developed independently of SWIM.

The protocol used by SWIM is simple and text-based, so it is even possible to interact with SWIM using a TELNET client. SWIM listens on port 4242 by default. Once a client (i.e., an external adaptation manager) opens a connection to this port, SWIM waits for a command sent by the client as a text line terminated with a newline character. If the command is not recognized or there is an error executing it, SWIM replies with a text line with the prefix `'error:'`, followed by an error message. If the command is an execution command, such as `'set_dimmer 0.4'`, and it succeeds, SWIM replies `'OK'`. If the command is a probing command, SWIM replies with a command-dependent answer (an integer or a floating-point number) in a complete line. For example, if the client sends the command `'get_dimmer'` after having sent the previous command, SWIM replies with `'0.4'`

Listing 2 shows the implementation of a simple external adaptation manager included with the artifact. In this case, the interaction with SWIM using its TCP-base interface is encapsulated in a class `SwimClient`.[6] This class has a method to open the connection to SWIM (e.g., `swim.connect("localhost")`), and when it is connected, the other methods use this connection to send the commands and wait for the response from SWIM. Unlike the adaptation manager implemented as a simulation model, this adaptation manager has to implement the complete self-adaptation loop—the while loop between lines 2–33. The first part of the code inside the loop implements the *monitor*, using the probing commands of the interface to get information about the system, with the local variables where the results are stored being the *model*. In the adaptation manager of Listing 1, there is a section of code similar to this, but in that case it was just for convenience, because SWIM has already updated the *model* in that case. Here, however, in addition to convenience it is a matter of performance, since every invocation of a method of `SwimClient` implies a round-trip request to SWIM. The rest of the code implements the same logic as the example in Section 3.1. The only difference is that instead of creating a `MacroTactic` to pass it on to the execution manager, this adaptation manager directly uses the effector commands of the SWIM interface, as it does not have a separate *execute* activity.

[6]This class can be easily reused to implement other adaptation managers, but that is not necessary.

```
1  void simpleAdaptationManager(SwimClient& swim) {
2      while (swim.isConnected()) {
3          double dimmer = swim.getDimmer();
4          int servers = swim.getServers();
5          int activeServers = swim.getActiveServers();
6          bool isServerBooting = (servers > activeServers);
7          double responseTime = swim.getAverageResponseTime();
8
9          if (responseTime > RT_THRESHOLD) {
10             if (!isServerBooting
11                     && servers < swim.getMaxServers()) {
12                 swim.addServer();
13             } else if (dimmer > 0.0) {
14                 dimmer = max(0.0, dimmer − DIMMER_STEP);
15                 swim.setDimmer(dimmer);
16             }
17         } else if (responseTime < RT_THRESHOLD) { // can we incr. dimmer or remove servers?
18
19             // only if there is more than one server of spare capacity
20             double spareUtilization = activeServers − swim.getTotalUtilization();
21
22             if (spareUtilization > 1) {
23                 if (dimmer < 1.0) {
24                     dimmer = min(1.0, dimmer + DIMMER_STEP);
25                     swim.setDimmer(dimmer);
26                 } else if (!isServerBooting && servers > 1) {
27                     swim.removeServer();
28                 }
29             }
30         }
31
32         sleep(PERIOD);
33     }
34 }
```

Listing 2: Adaptation manager as an external program.

4 EXPERIMENTS

Running an experiment with SWIM consists of three steps. First, the simulation has to be configured, setting the duration, the input trace, and the service time for processing requests, among other things. Second, the experiments have to be run, and how that is done depends on whether the adaptation manager is a simulation module or an external program. Finally, the results obtained are analyzed. The following sections describe these steps.

4.1 Configuring the Simulation

SWIM has two simulation directories under `swim/simulations`, one for each kind of adaptation manager. These directories are called `swim_sa` and `swim` for the adaptation manager as a simulation module and as an external program, respectively. A simulation is configured with an `.ini` file like the one shown in Listing 3. Here, we provide a brief overview of parts relevant to SWIM, but for more details about the syntax and other options, we refer the reader to the OMNeT++ manual [22]. A simulation must have at least one named *configuration* defined. This example defines a configuration named `Reactive` in line 51. In line 52, it specifies the module type name to be used for the adaptation manager. All the simulation parameters not defined by a configuration are taken from the `[General]` section, thus allowing defining multiple similar configurations without repeating configuration entries. The first part in this section (lines 2–17) contains boilerplate configuration for the recording of results. Line 19 specifies the network for the simulation, which is a specification of how the different simulation modules are connected. The `SWIM_SA` network is the one used when the adaptation manager is a simulation module.

```
1  [General]
2  num-rngs = 3
3
4  # save results in sqlite format
5  output-vector-file = ${resultdir}/${configname}-${runnumber}.vec
6  output-scalar-file = ${resultdir}/${configname}-${runnumber}.sca
7  outputscalarmanager-class = "omnetpp::envir::SqliteOutputScalarManager"
8  outputvectormanager-class = "omnetpp::envir::SqliteOutputVectorManager"
9
10 # non-default statistics recording
11 *.initialServers.param-record-as-scalar = true
12 *.maxServers.param-record-as-scalar = true
13 *.bootDelay.param-record-as-scalar = true
14 *.numberOfBrownoutLevels.param-record-as-scalar = true
15 *.evaluationPeriod.param-record-as-scalar = true
16 *.responseTimeThreshold.param-record-as-scalar = true
17 *.maxServiceRate.param-record-as-scalar = true
18
19 network = SWIM_SA
20 result-dir = ../../../results/SWIM_SA
21
22 # simulation input and duration
23 *.source.interArrivalsFile = ${trace = "traces/wc_day53-r0-105m-l70.delta",
          "traces/clarknet-http-105m-l70.delta"}
24 sim-time-limit = 6300s
25 warmup-period = 900s
26
27 # adaptation loop period
28 *.evaluationPeriod = 60
29
30 # adaptation manager params
31 *.numberOfBrownoutLevels = 5
32 *.responseTimeThreshold = 0.75s
33
34 # server pool configuration
35 *.maxServers = 3
36 *.initialServers = 3
37
38 # server config
39 **.server*.server.threads = 100
40 **.server*.server.timeout = 10s
41 **.server*.server.brownoutFactor = 0.1
42
43 # for plotting, use latency as iteration variable even if no iteration is needed
44 *.bootDelay = ${latency = 0, 60, 120, 180, 240} # deterministic boot times
45 #*.bootDelay = truncnormal( ${latency = 0, 60, 120, 180, 240}, ${stddev=($latency)/10} ) # random
          boot times
46
47 # service time configuration
48 **.server*.server.serviceTime = truncnormal(0.030s,0.030s)
49 **.server*.server.lowFidelityServiceTime = truncnormal(0.001s,0.001s)
50
51 [Config Reactive]
52 *.adaptationManagerType = "ReactiveAdaptationManager"
```

Listing 3: Simulation configuration file.

The user traffic to the applications is simulated by replaying request traces. The property `source.interArrivalsFile` (line 23) defines the path to the trace file. When a property is assigned multiple values, as in this case, it defines a (possibly named) iteration variable, allowing the definition of experiments that include multiple runs with different parameters. We will come back later to this when we introduce another iteration variable.

Line 24 specifies the duration of the simulation, which in this case matches the duration of the input trace. Line 25 defines a warm-up period of 900 seconds, during which the simulation runs normally, but no statistics are collected. This is useful for adaptation managers that, for example, have filters or estimators that require some priming.

The period of the self-adaptation loop is defined in line 28. Although this does not affect an adaptation manager running as an external program, it does determine the period with which statistics are recorded in the results. Lines 31–32 are parameters for the adaptation manager. Some adaptation managers use a discrete number of dimmer levels, and that number can be set here.[7] The `responseTimeThreshold` parameter defines the threshold below which the average response time must be kept.

The configuration of the pool of servers includes the maximum number of servers (line 35) and the number of servers active when the simulation starts (line 36). In addition, there are parameters that in a real deployment are typically found in the HTTP server software. These include the number of threads (i.e., concurrent requests) for each server (line 39), and the timeout for requests in the queue waiting to be served (line 40). Line 41 specifies the initial brownout factor for the servers. SWIM can also simulate the time it takes to boot a server or to provision one from the cloud. This is accomplished with the `bootDelay` parameter. In line 44, multiple values in seconds are assigned to this parameter, defining the iteration variable `latency`. OMNeT++ creates a simulation *run* for each unique assignment of all the iteration variables in a configuration,[8] with the last iteration variable defined being the one that iterates faster across runs. In this example, `trace` is the outer iteration variable with two values, and `latency` is the inner iteration variable with 5 values, and together they define 10 runs. Run 0 uses the first trace and `latency=0`, run 1 uses the first trace and `latency=60`, and so on. Run 5 is the first one to use the second trace because the runs 0 through 4 used all the values for the inner iteration variable. Therefore, run 5 uses the second trace and `latency=0`, run 6 uses the second trace and `latency=60`, and so on. In this case, the boot delays are deterministic, but it is also possible to have random boot delays. The example in line 45 indicates that the value for `bootDelay` has to be drawn from a normal distribution (truncated to the non-negative range) with mean equal to `latency` and standard deviation equal to one tenth of `latency`.

The final piece of configuration defines the service time for the two classes of responses (lines 48–49). The value of the parameter `server.serviceTime` is the service time for the responses that include the optional content, and `server.lowFidelityServiceTime` is for responses that do not include the optional content. The service time is the time it takes to process a request if there is no wait or contention when processing it. These values can be obtained by profiling requests processed in a real system making sure that only one request is sent at a time. In general, the service time is not constant but follows some probability distribution. In the example, a truncated normal distribution defined by the mean and standard deviation parameters is used.

4.2 Running Experiments

Running a simulation requires selecting which named configuration defined in the configuration file to use, and which run numbers to run. Each simulation directory provides a script called `run.sh` that accepts the following arguments.[9]

```
./run.sh config [run-number(s)|all [ini-file]]
```

The optional argument to select which run to execute can include a list of comma-separated run numbers without spaces (e.g., `1,3,4`), or a range or run numbers (e.g., `0-4`). The optional third argument

[7] *brownout* is used as the complement of *dimmer* (i.e., $dimmer = 1 - brownout$).
[8] Except when *parallel iteration* is used [22].
[9] Detailed instructions for running experiments are provided with the artifact.

can be used to specify a different `.ini` file (the default is `swim.ini` and `swim_sa.ini` in `swim` and `swim_sa` respectively).

When the simulation runs, it prints logging information, and after each run completes, the results are saved in the directory specified in the configuration file. The next section describe how to access and analyze the results.

4.3 Analyzing Results

SWIM is configured to save the results of the simulations in SQLite databases under the directory named `results`, which is in the same directory as `swim`. The results are saved in `results/SWIM` for the simulation `simulations/swim`, and in `results/SWIM_SA` for the simulation `simulations/swim_sa`. There are two results databases for each run named with the configuration name and the run number. One of the databases (`.sca`) has scalar values, such as the boot delay and the maximum number of servers, and the other (`.vec`) has vectors such as the number of active servers at different times, and the response time for each request.

There are several tools that work with SQLite databases that can be used to analyze the results. For example, it is possible to export the tables in the database to plain-text files with comma-separated values, or use the OMNeT++ IDE to plot charts with the results. More details, including the database schema for the results are provided by Hornig [11].

SWIM includes a tool to create plots with the results of the simulation. This tool is written in R, a language for statistical computing [23], and it provides an example of how to process results, which can be modified for other purposes. This tool can be invoked with `swim/tools/plotResults.sh` and can be used to generate plots in PDF or PNG format. Figure 2 shows the plot generated for one of the simulation runs. The top chart plots the average request rate per evaluation period, as defined in the configuration. The server charts show the number of active servers with a solid line, and the total number of servers (both active and booting) with a dashed line. The difference between the two shows when a server is booting. The chart in the middle shows the value of the dimmer at different times. In this case, the adaptation manager uses discrete dimmer levels that appear as steps in the plot, but it does not have to be that way. The average response time chart shows a dashed line representing the response time threshold. Finally, the utility chart at the bottom of Figure 2 plots the cumulative utility using the utility function used in a comparative study of predictive self-adaptation approaches [20]. The utility function used for this plot can be changed in `swim/tools/plotResults.R`.

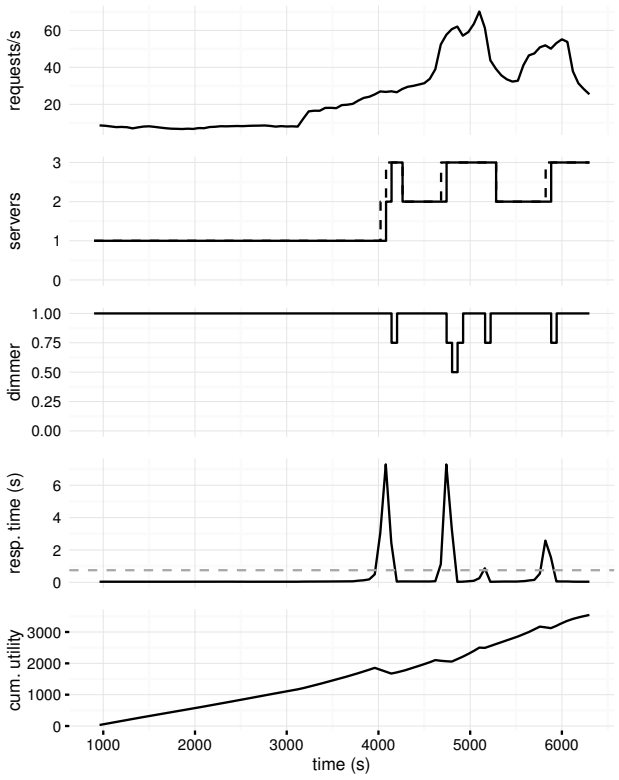

Figure 2: Generated plot of simulation results.

5 CONCLUSION

In this paper we have presented SWIM, a simulator of web applications. SWIM addresses several issues that researchers face when developing and evaluating self-adaptation approaches for web applications, namely the difficulty or cost of deploying multi-server applications, not being able to replicate conditions, and the time it takes to run experiments. SWIM can simulate tens of servers in a single computer without any special configuration other than specifying how many servers are needed. In addition, SWIM can exactly replicate run-time conditions between runs, enabling the comparison of different approaches. When used with a self-adaptation manager built as a simulation module, SWIM can simulate hours of web traffic in a few minutes. The exemplar is available as a Docker image (https://hub.docker.com/r/gabrielmoreno/swim/) for easy deployment on different platforms, and its source code is available at https://github.com/cps-sei/swim.

ACKNOWLEDGMENTS

Copyright 2018 ACM. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center, by AFRL and DARPA under agreement number FA8750-16-2-0042, by the Office of Naval Research under grants N000141612961 and N00014172889, and by the National Science Foundation under award CCF-1618220. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, ONR, DARPA, NSF, or the U.S. Government. DM18-0083

REFERENCES

[1] Konstantinos Angelopoulos, Alessandro V. Papadopoulos, Vitor E. Silva Souza, and John Mylopoulos. 2016. Model predictive control for software systems with

CobRA. In *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '16*. ACM Press, Austin, Texas, 35–46. https://doi.org/10.1145/2897053.2897054

[2] Martin F. Arlitt and T. Jin. 2000. A workload characterization study of the 1998 World Cup Web site. *IEEE Network* 14, 3 (2000), 30–37. https://doi.org/10.1109/65.844498

[3] Martin F. Arlitt and Carey L. Williamson. 1996. Web server workload characterization. *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '96* 24 (may 1996), 126–137. https://doi.org/10.1145/233013.233034

[4] Tao Chen and Rami Bahsoon. 2017. Self-Adaptive Trade-off Decision Making for Autoscaling Cloud-Based Services. *IEEE Transactions on Services Computing* 10, 4 (jul 2017), 618–632. https://doi.org/10.1109/TSC.2015.2499770

[5] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. 2009. Evaluating the effectiveness of the Rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 132–141. https://doi.org/10.1109/SEAMS.2009.5069082

[6] J Cito and H C Gall. 2016. Using Docker Containers to Improve Reproducibility in Software Engineering Research. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 906–907.

[7] Alessio Gambi, Mauro Pezze, and Giovanni Toffetti. 2016. Kriging-Based Self-Adaptive Cloud Controllers. *IEEE Transactions on Services Computing* 9, 3 (may 2016), 368–381. https://doi.org/10.1109/TSC.2015.2389236

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20%5C&path=ASIN/0201633612

[9] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. 2014. Adaptive, Model-driven Autoscaling for Cloud Applications. In *11th International Conference on Autonomic Computing*. 57–64. https://www.usenix.org/system/files/conference/icac14/icac14-paper-gandhi.pdf

[10] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (oct 2004), 46–54. https://doi.org/10.1109/MC.2004.175

[11] Rudolf Hornig. 2016. SQLite as a Result File Format in OMNeT++. (2016). https://summit.omnetpp.org/archive/2016/assets/pdf/OMNET-2016-Session_3-03-Presentation.pdf OMNeT++ Community Summit 2016.

[12] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. 2009. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Middleware 2009, ACM/IFIP/USENIX, 10th International Middleware Conference*, Jean Bacon and Brian F. Cooper (Eds.). Springer, Urbana, IL, 163—-183. http://link.springer.com/chapter/10.1007/978-3-642-10445-9_9

[13] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2009. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *Proceedings of the 6th international conference on Autonomic computing - ICAC '09*. ACM Press, New York, New York, USA, 117. https://doi.org/10.1145/1555228.1555261

[14] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055

[15] Malik Jahan Khan, Mian Muhammad Awais, Shafay Shamail, and Irfan Awan. 2011. An empirical study of modeling self-management capabilities in autonomic systems using case-based reasoning. *Simulation Modelling Practice and Theory* 19, 10 (nov 2011), 2256–2275. https://doi.org/10.1016/j.simpat.2011.08.005

[16] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. 2014. Brownout: building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM, New York, New York, USA, 700–711. https://doi.org/10.1145/2568225.2568227

[17] Markus Luckey, Benjamin Nagel, Christian Gerth, and Gregor Engels. 2011. Adapt Cases: Extending Use Cases for Adaptive Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM Press, New York, New York, USA, 30—-39. https://doi.org/10.1145/1988008.1988014

[18] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 1–12. https://doi.org/10.1145/2786805.2786853

[19] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2016. Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Wuerzburg, Germany, 147–156. https://doi.org/10.1109/ICAC.2016.59

[20] Gabriel A. Moreno, Alessandro Vittorio Papadopoulos, Konstantinos Angelopoulos, Javier Camara, and Bradley Schmerl. 2017. Comparing Model-Based Predictive Approaches to Self-Adaptation: CobRA and PLA. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 42–53. https://doi.org/10.1109/SEAMS.2017.2

[21] Gabriel A. Moreno, Ofer Strichman, Sagar Chaki, and Radislav Vaisman. 2017. Decision-Making with Cross-Entropy for Self-Adaptation. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 90–101. https://doi.org/10.1109/SEAMS.2017.7

[22] OMNeT++ 2018. OMNeT++ Simulation Manual. https://omnetpp.org/doc/omnetpp/manual/. (2018).

[23] R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

[24] Rahul Raheja, Shang-Wen Cheng, David Garlan, and Bradley Schmerl. 2010. Improving architecture-based self-adaptation using preemption. In *Self-Organizing Architectures*. Springer-Verlag, 21–37. http://dl.acm.org/citation.cfm?id=1880569.1880572

[25] RUBiS 2009. RUBiS: Rice University Bidding System. http://rubis.ow2.org/. (2009).

[26] Gabriel Tamura, Norha M Villegas, Hausi A Müller, Laurence Duchien, and Lionel Seinturier. 2013. Improving Context-awareness in Self-adaptation Using the DYNAMICO Reference Model. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '13)*. IEEE Press, Piscataway, NJ, USA, 153–162. http://dl.acm.org/citation.cfm?id=2663546.2663571

[27] András Varga and Rudolf Hornig. 2008. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems Workshop*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Marseille, France. http://dl.acm.org/citation.cfm?id=1416222.1416290

[28] Qiliang Yang, Jian Lü, Juelong Li, Xiaoxing Ma, Wei Song, and Yang Zou. 2010. Toward a fuzzy control-based approach to design of self-adaptive software. In *Proceedings of the Second Asia-Pacific Symposium on Internetware - Internetware '10*. ACM Press, New York, New York, USA, 1–4. https://doi.org/10.1145/2020723.2020738