Technische Universität Wien
Institute of Information Systems Engineering
Distributed Systems Group

FakeLoad: An Open-Source Load Generator

M. Sigwart, C. Hochreiner,
M. Borkowski, S. Schulte

msigwart@infosys.tuwien.ac.at

TUV-1942-2018-01 Nov. 14, 2018

In software testing, verification of a system's dynamic properties such as auto-scaling behavior can be vital for a project's success. However, testing such properties can be difficult when parts of the system are not implemented yet, mocked or simply not available, as those parts might have a great impact on the system's runtime behavior. To address this issue, we introduce FakeLoad, an open-source Java library capable of producing flexible, on-demand system load within applications or tests. Our evaluation shows that the library is able to produce the requested system loads with high accuracy and consistency.

Keywords: Cloud Computing, Elastic Processes, Optimization, Scheduling, Business Process Management

Argentinierstr. 8/194-2
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
http://www.infosys.tuwien.ac.at/

FakeLoad: An Open-Source Load Generator

Marten Sigwart, Christoph Hochreiner, Michael Borkowski, Stefan Schulte
Distributed Systems Group, TU Wien, Austria
{msigwart, c.hochreiner, m.borkowski, s.schulte}@infosys.tuwien.ac.at

Abstract—In software testing, verification of a system's dynamic properties such as auto-scaling behavior can be vital for a project's success. However, testing such properties can be difficult when parts of the system are not implemented yet, mocked or simply not available, as those parts might have a great impact on the system's runtime behavior. To address this issue, we introduce FakeLoad, an open-source Java library capable of producing flexible, on-demand system load within applications or tests. Our evaluation shows that the library is able to produce the requested system loads with high accuracy and consistency.

1 Introduction

In software development, an application's non-functional requirements are just as important as its functional requirements. Even though there exist many different definitions on what exactly non-functional requirements are [6], there is a general consensus that non-functional requirements describe *how a system works* as opposed to functional requirements that describe *what a system should do* [4], [11]. Examples for non-functional requirements are availability, scalability, performance, reliability, or robustness.

Non-functional requirements play an important role for a project's success. Therefore, their early verification is often vital. For instance, for a distributed system, we might want to test non-functional aspects like its auto-scaling behavior or its monitoring infrastructure. However, such dynamic properties can be hard to test. For instance, any method that performs a computationally complex or memory-intensive algorithm, or one that has extensive disk or network usage, might influence these properties. If these methods are not implemented yet, be it, because the algorithm has not left the research department, or the data access model is not clearly defined yet, testing performance or other dynamic non-functional requirements is difficult. When testing such an incomplete system, the developer is forced to make assumptions about how missing parts would influence the system, which yield additional risks. A developer might underestimate the real influence or overlook certain side effects of system components.

A solution to get a reliable estimate about the system's dynamic properties like auto-scaling behavior is to simulate system load instead of being forced to implement missing parts before being able to test. Tools like *consume.exe*, *cpustres.exe*, *HeavyLoad* [7], or *Diskspd* [2] for Windows or *stress* for Linux [14] can all be used to generate specific system load like CPU, memory or disk I/O. However, these tools are primarily used for stress testing purposes to test how an application or machine reacts to limited resources, they lack the possibility to generate load from within applications to simulate regular application behavior. This calls for a tool that can do exactly that: generate custom system load from within applications.

Not only does such a need arise in early-on testing of non-functional requirements such as auto-scaling behavior, it could also be required during testing where parts of the system are being mocked. It is a well-established practice in software testing to replace certain parts of a system with a mock [5]. A mock acts like the real component, however, it does not rely on its complex or complicated dependencies. Mocking frameworks like Mockito [9] or Easymock [3] can frequently be found among the most common Java dependencies [8].

However, mocking functionality of a system can sometimes have side effects when the mocked functionality does not actually reflect the real functionality. A tool that could generate system load like CPU, memory, disk IO, etc. in a simple way – on demand – could help create mocks that are close to the real object's behavior without inheriting any of the non-trivial code of the real object.

Last but not least, an on-demand load generating tool could also come in handy when dealing with algorithms or data which stand under a non-disclosure agreement (NDA). The terms of the NDA might prohibit the publication of any scientific evaluation involving the protected data or algorithm. A possible workaround could be to "simulate" the behavior of the protected data or algorithm, circumventing the NDA and thus allowing publication.

To tackle the issues stated above we introduce FakeLoad, an open-source Java library which lets us produce on-demand, flexible "fake" system load within our application or tests.

2 Design

Throughout this section, the main concepts and the general design goals of the FakeLoad library are introduced.

2.1 Design Goals

FakeLoad is designed as a library that can be used for generating "fake" system load within applications or tests.

The main design goals of the library can be summarized as follows:

- On-demand load generation
- Flexibility
- Accuracy
- Concurrency
- Easy-to-use Application Programming Interface (API)

2.1.1 On-demand load generation

As addressed in the introduction, common load generating tools like *stress* or *consume.exe* act as a separate entity from the actual application to generate system load. They lack the possibility to generate load from within applications. FakeLoad is designed to let developers generate system load *on demand*, i.e, the library offers the possibility to generate system load whenever needed, and from whichever code it is required (i.e., the application itself, or test code).

2.1.2 Flexibility

FakeLoad is designed to provide *flexible* load generating capabilities. Not only should developers be able to generate load whenever and wherever needed, they should also be able to specify exactly what kind of system load should be generated. To simulate the behavior of specific algorithms or operations, the library needs to offer the capability of specifying *load patterns*. For instance, a load pattern could look like the following: *Simulate a CPU load of 80% and a memory load of 1024 MB for 5 seconds then simulate a CPU load of 30% and a memory load of 2048 KB for 10 seconds.*

2.1.3 Accuracy

The library should generate system load as requested by the user as accurately as possible.

2.1.4 Concurrency

We require the library to be capable of *concurrent load simulation*. This means that two separate threads each requesting load simulation should both be able to do so without conflicting each other. In more concrete terms, if one thread is requsted to simulate a CPU load of 50% and another thread is requested to simulate a CPU load of 30%, the total load created by the library should be 80%. Not only should the library support concurrent execution, it should also clearly define its behavior in edge cases. For instance, the behavior in case the overall requested load is close to, or in excess of 100%, should be clearly defined and documented. Concurrency behavior of the library is discussed in Section 3.4.

2.1.5 Easy-to-use API

Besides functionality, usability is the most important factor for the success of an API [10]. If an API does not behave as documented, it naturally offers no value to the user. However, if an API is not easy to use, it might remain unused despite functional completeness. Among FakeLoad's design goals is the ease of use of the resulting API. FakeLoad is primarily designed as a support library for testing. Developers should be able to use FakeLoad effectively and with as little learning and training as possible. Naturally, an easy-to-use library also means providing plentiful and useful material, such as tutorials or in-code documentation. In the best case, developers should be able use FakeLoad efficiently in a "copy and paste" kind of manner by merely reading the introductory tutorial.

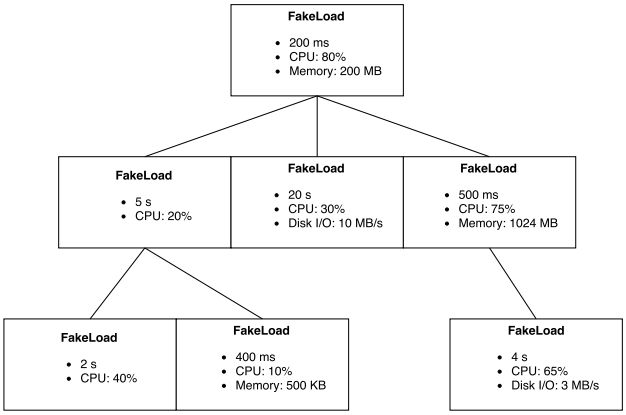

Fig. 1: FakeLoad tree

2.2 Core Concepts

The library is designed around two main concepts: A *FakeLoad* and a *FakeLoadExecutor*. As the names suggests, a FakeLoad represents artificial (requested) system load, and the FakeLoadExecutor is responsible for actually creating the requested load.

2.2.1 FakeLoad

FakeLoad is one of the two core classes of the FakeLoad library. A FakeLoad object contains the instructions needed for load generation. For instance, for simulating a CPU load of 50%, we create a FakeLoad object containing CPU load instructions of 50%. A FakeLoad object can contain three different kinds of system load:

- CPU
- Memory
- Disk Input/Output

Besides specifying the kinds of system load (CPU, memory, Disk I/O), users can also define the duration and repetitions of a FakeLoad object. The duration defines for how long the specified system load are executed and the number of repetitions defines how many times the load defined in a given FakeLoad object will be repeated during execution.

For a more flexible system load configuration, a FakeLoad object can contain other FakeLoad objects. These inner FakeLoad objects can be used to define more elaborate load patterns. Users can add as many inner FakeLoad objects as they want and these inner FakeLoad objects themselves can again contain other FakeLoad objects. This results in a tree structure, as demonstrated in Figure 1.

A FakeLoad object does not define any formal requirements on the order of execution. It is up to the FakeLoadExecutor to decide in which order the different FakeLoad object nodes are processed. However, a FakeLoad object should offer some kind of default order that the executor can use (see Section 3.2.3).

2.3 FakeLoadExecutor

The FakeLoadExecutor is the second core class of the FakeLoad library. While a FakeLoad object is used to specify the system load a user requires to generate, the FakeLoadExecutor is primarily concerned with how system load is actually created. Users therefore execute FakeLoad objects using a FakeLoadExecutor. The executor is responsible for generating system load as specified by the FakeLoad object submitted by the user. The executor is also responsible for parsing the tree of a FakeLoad object as seen in Figure 1. By construction, the executor defines the order in which nodes are processed.

Why a separate executor?

In first library designs, the two concepts *FakeLoad* and *FakeLoadExecutor* were united in a single entity. Clients would use a FakeLoad object to specify the system load instructions as well as actually executing those instructions. The rationale was to keep client concerns to a minimum by making the interface as simple as possible. By uniting the concepts of the FakeLoad and the FakeLoadExecutor, the main client interface consisted only of a single class. However, multiple reasons led to the separation into FakeLoad and FakeLoadExecutor: First, having one component for load specification (FakeLoad) and one for execution (FakeLoadExecutor) ensures a clear separation of concerns, which is usually desirable in software design as it improves testability and maintainability. Secondly, the split allowed us to define the FakeLoad as *immutable value object* which came with several advantages. The split increases the flexibility in terms of FakeLoad execution. Instead of simulation behavior being hard-coded into the FakeLoad component, clients are able to simulate the same FakeLoad obejcts using different types of FakeLoadExecutors. Furthermore, this enables users to create executors tailored specifically for their needs. For example, a user might need to simulate numerous kinds of system load types in which case a complex multi-threaded executor is needed. In another case, a user might only want to simulate memory usage, in which case a simple single threaded executor that would only allocate some memory is sufficient.

3 Implementation

In this section, the main API and important implementation details of the FakeLoad library are addressed. Note that throughout this section, to emphasize when we are referring to concrete types of the library we will use `monospaced` typesetting. For example, `FakeLoad` refers to the concrete library class whereas when the standard font is used, FakeLoad refers to the general concept.

3.1 The Main API

As described in Section 2, the library consists of two main classes: *FakeLoad* and *FakeLoadExecutor*. A FakeLoad object *contains* load instructions, and a FakeLoadExecutor *executes* those instructions. The main API of the library reflects these two concepts pretty well. For example, Listing 1 below shows how to create and execute a FakeLoad that will simulate a CPU load of 80% and a memory load of 300 megabytes for ten seconds:

As we can see, the user first creates a `FakeLoad` and fills it with different system load instructions. Then, the user executes the `FakeLoad` object by creating a `FakeLoadExecutor` and calling its `execute()` method passing the newly created FakeLoad object as a parameter. With only these two concepts, any combination of system load can be created.

```
// Creation
FakeLoad fakeload = FakeLoads.create()
        .lasting(10, TimeUnit.SECONDS)
        .withCpu(80)
        .withMemory(300, MemoryUnit.MB);

// Execution
FakeLoadExecutor executor =
    FakeLoadExecutors.newDefaultExecutor();
executor.execute(fakeload);
```

Listing 1: FakeLoad creation and execution

3.2 FakeLoad

Section 2.2.1 introduced FakeLoad as one of the two main classes of the library. In the library, a FakeLoad object is represented by interface `FakeLoad`. Within a `FakeLoad` object, users can specify which kinds of system load they want to simulate and for how long they will be simulated. `FakeLoad` objects can be created either using a factory or a builder.

3.2.1 FakeLoad Creation

We already got a glimpse of how users can create `FakeLoad` instances in Listing 2. We see that FakeLoad creation follows a fluent interface. After creation, different load instructions like CPU, memory, disk I/O or other FakeLoad objects can be added by chaining the corresponding methods. For example, Listing 2 shows how the FakeLoad tree from Figure 1 could be created.

Users can not directly call a constructor to create `FakeLoad` instances. Instead, the factory class `FakeLoads` is used. This way, implementation details remain hidden as potential users do not need to know how exactly a FakeLoad object is implemented behind the covers. The use of a factory has another advantage as it leaves the option of being able to offer different ways of FakeLoad creation in the future versions of the library. For example, it could be convenient to store FakeLoad definitions as JSON strings. In this case, another method could be added to the factory, which would create `FakeLoad` instances from JSON strings.

```
// Complex FakeLoad creation
FakeLoad fakeload = FakeLoads.create()
    .lasting(200, TimeUnit.MILLISECONDS)
    .withCpu(80)
    .withMemory(200, MemoryUnit.MB)
    .addLoad(FakeLoads.create()
        .lasting(5, TimeUnit.SECONDS)
        .withCpu(20)
        .addLoad(FakeLoads.create()
            .lasting(2, TimeUnit.SECONDS)
            .withCpu(40))
        .addLoad(FakeLoads.create()
            lasting(400, TimeUnit.MILLISECONDS)
            .withCpu(10)
            .withMemory(500, MemoryUnit.KB)))
    .addLoad(FakeLoads.create()
        lasting(500, TimeUnit.MILLISECONDS)
        .withCPU(75)
        .withMemory(1024, MemoryUnit.MB)
        .addLoad(FakeLoads.create()
            lasting(4, TimeUnit.SECONDS)
            .withCpu(65)
            .withDiskOutput(30, MemoryUnit.KB)));
```

Listing 2: Complex FakeLoad creation

Besides the factory, users can also use the builder pattern to create FakeLoad instances. A builder class FakeLoadBuilder is provided for less expensive FakeLoad creation which could come in handy when very complex FakeLoad objects are created. This is due to the immutability of the FakeLoad class. Listing 3 shows a simple example of how FakeLoad objects are created using the builder class.

MemoryUnit: As we have shown in Listing 2 and 3, load instructions are mostly specified as numerical values [13]. However, we note that for specifying memory and disk I/O, we also pass an instance of MemoryUnit enum to the corresponding methods. The MemoryUnit enum eases the creation of memory-related system load. It entails the most common memory representations bytes, kilobytes (kB), megabytes (MB), and gigabytes (GB).

The orders of magnitude between bytes, kilobytes, megabytes, and gigabytes are represented aligned to powers of two, i.e., one kilobyte are 1024, one megabyte are 1024^2 and one gigabyte are 1024^3 bytes. These representations are sometimes also known as kibibytes (kiB), mebibytes (MiB) and gibibytes (GiB).

Immutable vs. Mutable

All classes implementing the FakeLoad interface are immutable. Immutability implies that an object cannot be modified after it has been created. The only way to work with an existing object is to create a new one based on the old one with updated values. Immutable objects are usually preferred when dealing with so-called *value objects*. Value objects are used for objects representing exactly one value, such as location, money, or time. A FakeLoad object can be considered a value object because it represents exactly one kind of load configuration. Also, FakeLoad objects containing exactly the same load configuration are considered equal. Making the FakeLoad classes immutable yields a couple of advantages with comparable few disadvantages for the overall architecture of the library.
The primary advantage of making FakeLoad classes immutable is the built-in thread safety. Immutable objects are inherently thread-safe, as there is no state which requires cross-thread synchronisation. This mitigates the problem of inconsistent state. Another advantage is that no checks for recursiveness is required, i.e., whether a FakeLoad object is about to add itself to its children nodes. Adding a FakeLoad object to itself would cause a StackOverflowError if the object were mutable. This is not possible with immutable objects as the adding of a FakeLoad to itself would cause a new FakeLoad object to be created containing the added object.
The only real disadvantage is slightly higher cost at object creation. The fluent interface of an immutable FakeLoad requires the creation of a new FakeLoad object whenever a load instruction is added or modified. However, this disadvantage could be overcome by either using the builder pattern (which comes at a slight cost of readability) or using an internal mutable object. Furthermore, because of the create-and-execute nature of a FakeLoad object, it can be assumed that an existing object will not be modified repeatedly. Therefore, the performance impact is negligible for most scenarios. However, just in case huge FakeLoad objects are required, the library provides the builder class FakeLoadBuilder.

3.2.2 Class Hierarchy

The interactions with FakeLoad objects are defined in the FakeLoad interface. The interface is backed by two concrete implementing classes. The factory and builder classes return one of the two implementing classes depending on the parameters passed during object creation. The UML class diagram of the FakeLoad classes is shown in Figure 2.

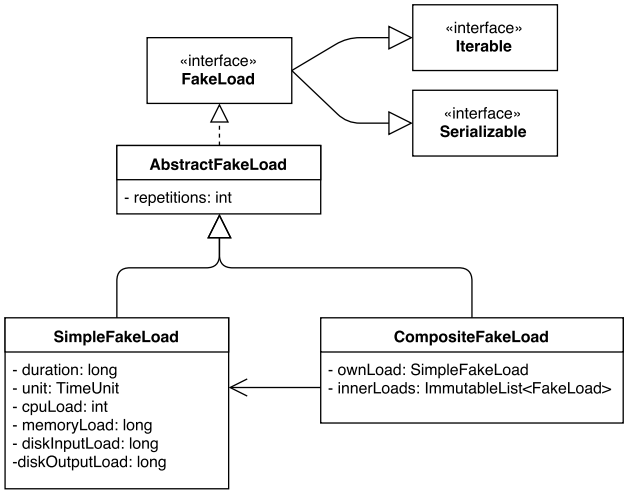

Fig. 2: UML class diagram of FakeLoad classes

3.2.2.1 SimpleFakeLoad: The SimpleFakeLoad class, as the name suggests, is responsible for simple load instructions. This means that a SimpleFakeLoad object contains fields which specify different types of system load, the duration as well as the number of repetitions. A SimpleFakeLoad does *not contain* inner FakeLoad objects.

3.2.2.2 CompositeFakeLoad: On the other hand, the CompositeFakeLoad class is a composite of a SimpleFakeLoad object specifying its own load instructions and multiple inner FakeLoad objects. When dealing with load patterns involving multiple inner FakeLoad objects, the underlying class is always CompositeFakeLoad.

```
// FakeLoad creation using the Builder pattern
FakeLoad fakeload = new FakeLoadBuilder(20,
    TimeUnit.SECONDS)
    .withCpu(20).withMemory(100, MemoryUnit.KB)
    .addLoad(new FakeLoadBuilder(20,
        TimeUnit.SECONDS)
        .withCpu(40).withMemory(200,
            MemoryUnit.KB).build())
    .addLoad(new FakeLoadBuilder(20,
        TimeUnit.SECONDS)
        .withCpu(60).withMemory(300,
            MemoryUnit.KB).build())
    .build();
```

Listing 3: FakeLoad creation using a builder

3.2.3 Default Execution Order (Iterator)

The FakeLoad interface extends the Iterable interface, each concrete class therefore implements the iterator() method. The returned iterator plays an important role for the execution of a FakeLoad object. In Section 2.2.1, we state that a FakeLoad object should offer a default order for executing the tree of its FakeLoad child nodes. The returned iterator serves this purpose by iterating over all the nodes of the tree, thus defining a default execution order of a FakeLoad object. The iterator is implemented in the following fashion: The iterator for the SimpleFakeLoad class returns only the SimpleFakeLoad object itself. The iterator for the CompositeFakeLoad class first returns its own SimpleFakeLoad object, and then calls the iterators of its children FakeLoads in the order in which they were added to the parent.

Serializable

The FakeLoad interface also extends the Serializable interface. This is done to allow sending or receiving FakeLoad objects over the network, or writing or reading them to or from a file. However, the SimpleFakeLoad and CompositeFakeLoad classes do not use the default serialization process of Java, but rather make use of the Serialization Proxy Pattern which was first defined in the book Effective Java by Joshua Bloch (2nd edition: item 78) [1]. The serialization proxy is useful as it hides implementation details of the classes. Further, it is also useful for keeping class invariants intact, because objects are recreated after deserialization by using the public API of the library.

3.3 FakeLoadExecutor

As described in Section 2.3 the FakeLoadExecutor is the second core concept of the FakeLoad library and is responsible for executing FakeLoads. In the library, this concept is represented by the FakeLoadExecutor interface. Similar to the process for creating FakeLoad objects, a factory class FakeLoadExecutors is available for creating FakeLoadExecutor instances. In Listing 4, we show how a client obtains a FakeLoadExecutor instance using the factory.

```
FakeLoadExecutor executor =
    FakeLoadExecutors.newDefaultExecutor();
    executor.execute(fakeload);
```

Listing 4: Creation of the default FakeLoadExecutor

This method returns a DefaultFakeLoadExecutor instance which, in the current version, is the only FakeLoadExecutor implementation available. Other kinds of executors can be implemented in future versions of the library or by users of the library themselves. When executing FakeLoads concurrently, clients are advised to always create FakeLoadExecutor instances using the factory method.

3.3.1 DefaultFakeLoadExecutor

The default executor is the preferred way for executing FakeLoad objects, as it already provides the capability necessary for simulating CPU, memory, and disk I/O. The default executor is capable of executing multiple FakeLoad objects concurrently, i.e. if one thread executes a CPU load of 30% and another thread executes a CPU load of 40%, the total CPU load simulated will be 70%. When the executor's execute(FakeLoad) method is called, it blocks until the execution of the passed FakeLoad object finished successfully or throws an exception. Figure 3 shows the general process when executing FakeLoad objects with the DefaultFakeLoadExecutor. Once a client submits a FakeLoad object via the executor's execute(FakeLoad) method, the object is propagated to a FakeLoadScheduler instance. The scheduler is responsible for scheduling increases/decreases of system load at the correct time. Increases and decreases of system load are propagated to a specialized simulation infrastructure represented by an instance of the SimulationInfrastructure class. The infrastructure is responsible for the actual system load enactment, i.e., producing the desired system load like CPU percentage, amount of memory, or bytes/kilobytes/megabytes/gigabytes per second of disk I/O.

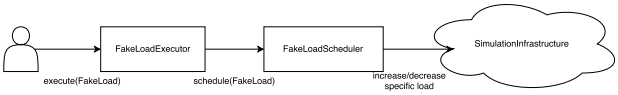

Fig. 3: General Process of FakeLoad execution

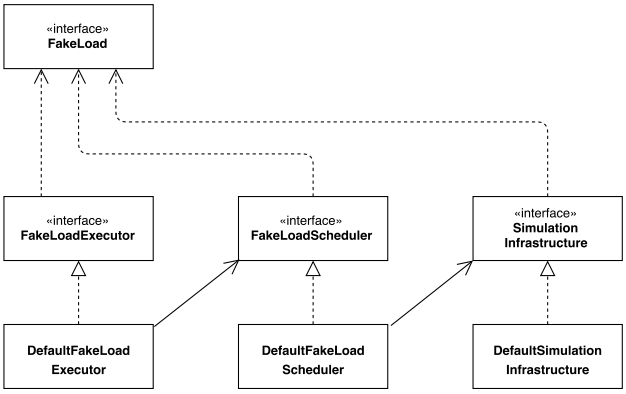

Fig. 4: UML class diagram of DefaultFakeLoadExecutor

As described above, in order to achieve correct concurrent load simulation behavior, especially when FakeLoads are executed from multiple threads, it is recommended to always create FakeLoadExecutor instances using the factory methods. All FakeLoadExecutor instances returned by the factory use the same SimulationInfrastructure internally. However, to allow customization by the client, clients can also manually create instances of the DefaultFakeLoadExecutor class using the constructor and injecting FakeLoadScheduler and SimulationInfrastructure instances. Prior to manually wiring up DefaultFakeLoadExecutor instances, clients should be aware of the inner workings of the class, in order to not cause incorrect concurrent behavior. Clients manually wiring up instances should also consider always passing the same SimulationInfrastructure instance. The UML class diagram of DefaultFakeLoadExecutor and its dependencies can be seen in Figure 4. More details on FakeLoadExecutor and corresponding classes can be found in the javadocs [13].

3.3.2 FakeLoadScheduler

The DefaultFakeLoadExecutor class depends on a FakeLoadScheduler instance for scheduling the execution of FakeLoads. The scheduling happens through a call to the schedule(FakeLoad) method of the scheduler. This method schedules the instructions contained in the FakeLoad object and passes them on to the simulation infrastructure at the right time and in the right order. The default FakeLoadScheduler implementation, realized in the DefaultFakeLoadScheduler class, uses the FakeLoad iterator to retrieve the next FakeLoad object in the chain of execution. For each FakeLoad object returned by the iterator, two tasks are executed. One task increases the system load, another task first pauses the current thread for the duration specified in the corresponding FakeLoad object and then decreases the system load. Both tasks are run asynchronously on one CompletableFuture [?] object,

effectively creating a chain of alternating increase and decrease tasks. If one of the tasks encounters an exception, execution of all remaining tasks is cancelled and the simulation of system load is interrupted. The exception is then propagated to the `DefaultFakeLoadExecutor`. More details on the `DefaultFakeLoadScheduler` can be found in the javadocs [13].

3.3.3 SimulationInfrastructure

The actual execution of system loads happens on a so-called simulation infrastructure. The simulation infrastructure is represented by the `SimulationInfrastructure` interface and its default implementation `DefaultSimulationInfrastructure`. The default implementation consists of multiple threads each responsible for simulating some specific system load, multiple threads for CPU simulation, one for memory simulation, and so on.

The `DefaultSimulationInfrastructure` class is responsible for starting and stopping these threads as well as managing the currently executed system load. The class contains the logic for aggregating all load instructions taken from FakeLoad objects passed to the infrastructure. Therefore, FakeLoad objects which are executed from different threads are aggregated in this class. In consequence, when using the DefaultSimulationInfrastructure class as infrastructure for simulating system load, only one instance should exist throughout the application to achieve correct concurrent behavior.

All threads are run as daemon threads in the background using a fixed thread pool. The use of daemon threads has the advantage that clients do not have to explicitly shut down the simulation infrastructure when the client program finishes, instead, all daemon threads are shut down when the Java Virtual Machine (JVM) exits. The following threads are part of the default infrastructure:

3.3.3.1 CpuSimulator: As the name suggests, the `CpuSimulator` class is responsible for generating CPU load during FakeLoad execution. CPU simulation is achieved through dividing a 100 ms time window between calculation and sleeping time. For instance, if the client requests a CPU load of 30%, the thread will perform calculations for 30 ms and then sleep for the remaining 70 ms. In the simulation infrastructure, there are as many `CpuSimulator` threads as the number of CPU cores which are available to the JVM.

3.3.3.2 MemorySimulator: The `MemorySimulator` class is responsible for utilizing memory during FakeLoad execution. Memory is utilized by creating byte arrays the size of the requested memory. Memory can be utilized in bytes/kilobytes/megabytes/gigabytes. Once byte arrays are created, the `MemorySimulator` thread waits until new instructions are available. When memory load is decreased again, the allocated byte arrays are cleared and replaced with a null reference to be made available for garbage collection.

3.3.3.3 DiskInputSimulator: The `DiskInputSimulator` is responsible for enacting disk input. Disk input can be defined as bytes/kilobytes/megabytes/gigabytes per second. Disk input is realized by dividing a time window of one second between reading the desired amount of data from a file on the hard drive and sleeping for the remainder of the second, thus achieving a disk input of about the desired amount per second. Naturally, the speed cannot exceed the maximum writing speed of the system.

In its current implementation, the `DiskInputSimulator` requires the file it uses for reading to be called *input.tmp* and be placed in the temporary directory as indicated by Java system property "java.io.tmpdir". The property can be set using the parameter `-Djava.io.tmpdir=/tmp`.

To prevent the computer from caching read data, the file used for reading should be at least twice the size as the RAM available.

3.3.3.4 DiskOutputSimulator: The `DiskOutputSimulator` is responsible for enacting disk output. Disk output can be defined as bytes/kilobytes/megabytes/gigabytes per second. Disk output is realized by splitting up the time window of one second between writing data to a file on the hard drive and sleeping for the remainder of the second. Thus, the average disk output will be about the desired amount per seconds. Again, the speed cannot exceed the maximum writing speed of the system.

Similar to the `DiskInputSimulator`, the `DiskOutputSimulator` writes to a file called *output.tmp* which will be placed in the temporary directory as indicated by Java system property `java.io.tmpdir`.

3.3.3.5 LoadControl: The `LoadControl` thread is responsible for controlling the system loads produced by other threads. For example, CPU simulation might not be accurate by solely splitting up 100 ms between operating and sleeping time, as other processes might be running that consume CPU. The `LoadControl` thread checks in regular intervals whether the generated system load is in proximity to the requested load and adjusts load generation if necessary. The default control interval is two seconds and the allowed margin for load generation is 1% off of the desired amount.

3.4 Concurrent Behavior

One of the design requirements of the FakeLoad library is to support concurrent FakeLoad execution. Concurrent behavior of the FakeLoad Library is defined as follows: When FakeLoads are executed concurrently from different threads, the load instructions of both FakeLoad objects get aggregated. That means, if one thread submits a CPU load of 20% and another thrad submits a CPU load of 30%, the total CPU load simulated by the API must be 50%.

When the total load accumulated by all requests exceeds the limit of the system, a runtime exception is thrown to notify the client. For instance, a CPU load of more than 100% is not possible. Therefore, when a client submits a FakeLoad object for execution which will cause a CPU load of more than 100%, the execute method of the `FakeLoadExecutor` will throw a runtime exception.

Exceeding of the memory limit is handled in a slightly different way: Even though the simulator thread responsible for simulating memory can check if enough memory is available for simulation, when actually allocating memory, the thread might still encounter an OutOfMemoryError. As the corresponding thread will die in such a case, the

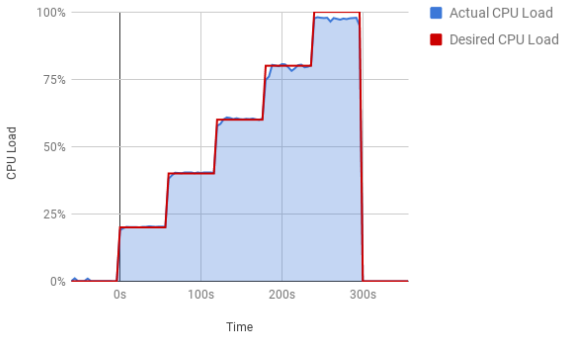

Fig. 5: Increasing CPU load

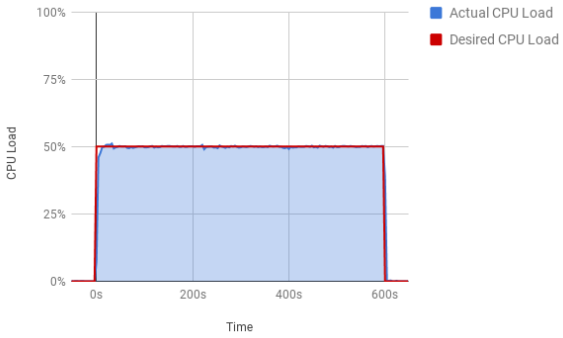

Fig. 6: Constant CPU load

executor starts a new thread and the requested memory is not executed. However, no runtime exception is thrown, the failure of memory simulation is only visible in the logs, other loads requested by the client are executed as usual. It is currently not possible to also exit execution with a runtime exception in case of an OutOfMemoryError, as communication between executor and simulation infrastructure is currently unidirectional, and reporting from infrastructure to executor is not supported.

Alternative Strategies

Besides aggregation of FakeLoad objects, other concurrent strategies could be useful. These alternatives are shortly addressed here and might be supported and configurable in future versions of the library.
Alternative 1: Last Come, First Served
The FakeLoad that is executed last gets prioritization. That means if one thread issues a CPU load of 20% before another thread issues a CPU load of 50%, the API simulates the 20% until the second request arrives, after which the library will produce 50% CPU.
Alternative 2: One at a Time Only one load request at a time is possible. That means if the library is currently executing a load request from one thread, any load requests issued by other threads will fail. This solution would virtually prevent the library's concurrent execution capabilities.

4 Evaluation

4.1 CPU

To evaluate the accuracy of the generated CPU load, we execute a FakeLoad object with a CPU load of 20%, 40%, 80% and 100%, each for a duration of 60 seconds, as well as a FakeLoad object with a constant CPU load of 50% for a duration of ten minutes. The actual CPU usage of the process was recorded with JConsole [12].

As we show in Figure 5, the FakeLoad library is able to produce a constant CPU load fairly close to the desired load levels. After a small adjustment period, every time a load change happens, the library adopts the load fairly quickly to be constantly within 1% of the desired load level. Especially for load levels under 80%, the library produces a very constant CPU load with hardly any fluctuations apart from the small adjustment periods in the beginning of each new load level. For CPU load levels of 80% and more, we see some more fluctuation. This fluctuation can be explained with other processes which are also competing for CPU time. At low load levels, other processes don't affect each other's CPU usage as much, as there is enough CPU power available. Logically, the library cannot reach an actual CPU usage of 100% because certain CPU power is required for kernel and system processes.

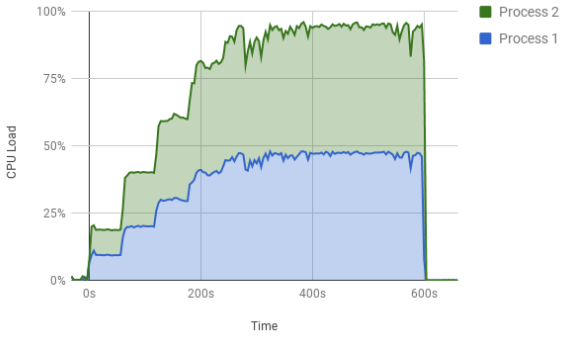

Fig. 7: Concurrent CPU simulation in two processes

To show that the library is also able to produce a constant CPU load over longer periods of time we execute a FakeLoad object with 50% CPU load for ten minutes. As we show in Figure 6, the FakeLoad library produces a CPU load which is very close (within 1%) to the desired 50% and stays constant over the entire simulation period of ten minutes.

Next, we execute two FakeLoad objects concurrently in different processes to establish how two different processes executing FakeLoad objects are influencing each other. Both FakeLoad objects are executed with rising CPU loads of 10%, 20%, 30%, ..., 100%, each executed for a minute. Both processes are started within a very short time of each other. The combined CPU measurements of both processes are shown in Figure 7. The graphs show clearly that both processes produce constant CPU loads of 10, 20, and 30%. The 40% loads are also produced by both processes even though we see some disturbances. Once both processes ought to simulate loads of 50% and more, they are no longer able to maintain the requested load. Both graphs show a trembling curve as the two processes compete with each other for CPU time. We also observe that no process seems to get prioritized, and both processes remain at around 47-48% of CPU usage.

We have shown that the library is able to produce accurate and consistent CPU loads. One thing to note when looking at these graphs, however, is that the loads represented

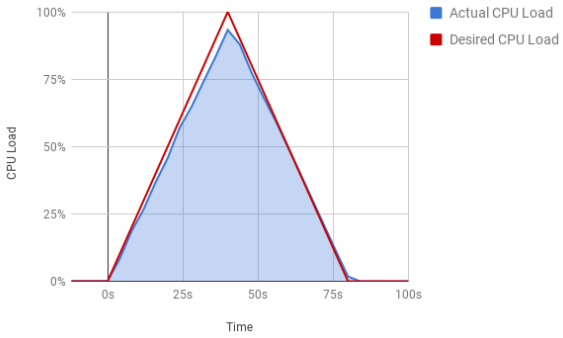

Fig. 8: CPU simulation with 4 second periods

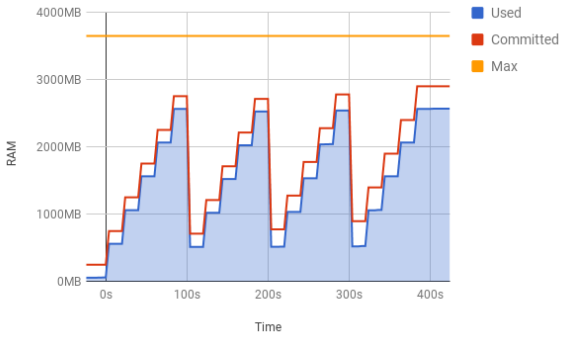

Fig. 9: Memory Usage

in the graphs are simulated for longer periods of time, where the library has enough time to adjust the actual load level according to the desired load level. When executing FakeLoad objects for only a couple of seconds or even just in the range of milliseconds, the CPU load produced will not be as accurate (see Figure 8).

4.2 Memory

The FakeLoad library simulates RAM usage by allocating the requested amount of bytes. To demonstrate the behavior of the library, we execute a FakeLoad object repeatingly allocating 500 MB of memory every 20 seconds. In Figure 9, we show the memory footprint of said execution. As we can see, every 20 seconds, 500 MB of memory are allocated. One might expect the memory graph to remain constant, however, the resulting graph is characterized by steps of 500 MB. In any case, we see that once the memory reaches a threshold, garbage collection kicks in and cleans up the unused memory leaving only about 500 MB.

4.3 Disk Input

Users specify disk input in FakeLoad objects in bytes per second. The library then simulates disk input by reading the requested amount of bytes from a file every second. To demonstrate disk input behavior of the library, we measure the execution of a FakeLoad object with increasing amounts of disk input. Disk input is measured with the iotop utility using a sampling rate of 1 second. The first FakeLoad object executed contains disk input loads of 10, 20, 30, 40, 50,

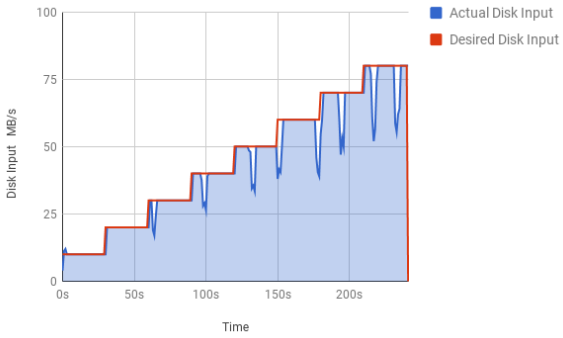

Fig. 10: Small increasing disk input loads

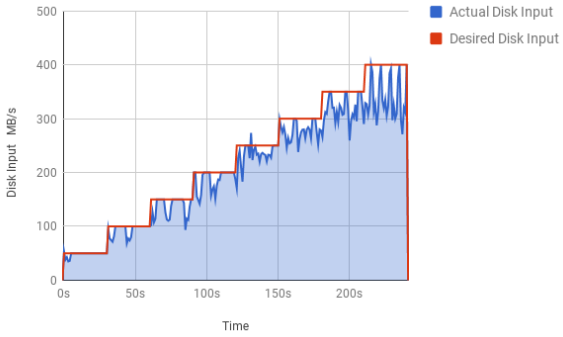

Fig. 11: Big increasing disk input loads

60, 70, and 80 MB/s, each executed for a duration of 30 seconds. The second FakeLoad object contains loads of 50, 100, 150, 200, 250, 300, 350, and 400 MB/s, also executed for 30 seconds each. As we can see in Figure 10, the library is able to produce the smaller loads in a consistent matter with a low number of outliers visible. However, in Figure 11, we observe that as loads become bigger, execution becomes less consistent, as the overall read limit of the system is reached.

To show that the library is also capable of producing consistent loads over longer periods of time, we execute a FakeLoad object with a disk input load of 50 MB/s for 10 minutes. In Figure 12 we show the results of the corresponding measurement.

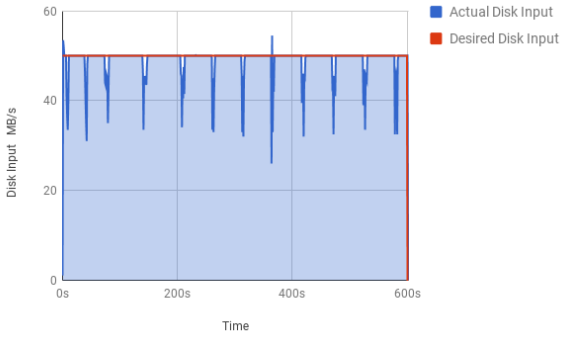

Fig. 12: Constant disk input load of 50 MB/s

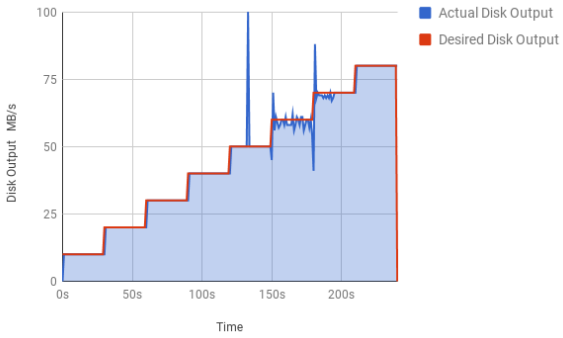

Fig. 13: Small increasing disk output

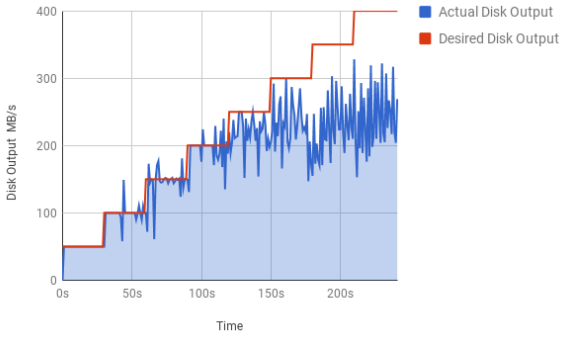

Fig. 14: Big Increasing Disk Output

4.4 Disk Output

In the same way as for disk input, disk output is specified in bytes per second by users of the FakeLoad library. The disk output is then enacted by writing the requested amounts of bytes every second to a file. Disk output was also evaluated by measuring the execution of different FakeLoad objects containing different disk output loads. Yet again, we used iotop with a sampling rate of 1 second as the monitoring tool. In Figure 13 and 14, we see the behavior of the library during small and big increasing disk output loads. We observe that smaller loads are handled in a precise and consistent manner. Bigger loads, on the other hand, are only reliably simulated up to limit of 200 MB/s, which suggests that the system's overall writing speed limit lies within that region.

To demonstrate that the library is able to produce constant disk output over longer periods of time, we also execute a FakeLoad containing 50 MB/s disk output load for 10 minutes. The results of the measurement is shown in Figure 15.

Some of the disturbance in the graphs is caused by the sampling rate of iotop. For example, while simulating a disk output of 50 MB/s, one sample might record a load 40 MB/s while the next sample records a load of 60 MB/s. This causes a disturbance in the graphs, however, averaging the values reveals an actual load of 50 MB/s.

5 Conclusion

This report introduced and discussed FakeLoad, a Java library for producing system loads within applications or

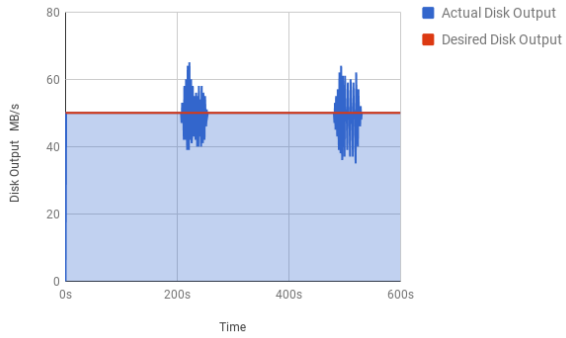

Fig. 15: Constant disk output load of 50 MB/s

tests. We showed that the library is able to generate constant, accurate and flexible system loads of CPU, memory, and disk I/O, even when generating system loads from multiple processes concurrently. However, possible future extensions exists, as especially for shorter simulation periods, the library is not able to produce loads as accurately as for longer simulation periods, due to the granularity for CPU and disk I/O load generation of 100 millisecond and 1 second, respectively.

References

[1] Joshua Bloch. *Effective java*. Pearson Education India, 2008.
[2] Diskspd. Technet diskspd utility: A robust storage testing tool. https://gallery.technet.microsoft.com/DiskSpd-a-robust-storage-6cd2f223. Retrieved August 12, 2017.
[3] EasyMock. Easymock. http://easymock.org/. Retrieved August 12, 2017.
[4] Ulf Eriksson. Functional requirements vs non functional requirements. http://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/, 2012. Retrieved August 12, 2017.
[5] Martin Fowler. Mocks aren't stubs. https://martinfowler.com/articles/mocksArentStubs.html. Retrieved August 12, 2017.
[6] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference*, pages 21–26. IEEE, 2007.
[7] HeavyLoad. Heavyload - free stress test tool for your pc. http://www.jam-software.com/heavyload/. Retrieved August 12, 2017.
[8] Henn Idan. The top 100 java libraries in 2016 - after analyzing 47,251 dependencies. http://blog.takipi.com/the-top-100-java-libraries-in-2016-after-analyzing-47251-dependencies/, 2016. Retrieved August 12, 2017.
[9] Mockito. Mockito framework site. http://site.mockito.org/. Retrieved August 12, 2017.
[10] Brad A Myers and Jeffrey Stylos. Improving api usability. *Communications of the ACM*, 59(6):62–69, 2016.
[11] NA. What is functional and non functional requirement? https://stackoverflow.com/questions/16475979/what-is-functional-and-non-functional-requirement. Retrieved August 12, 2017.
[12] Oracle. Jconsole. http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html/. Retrieved August 12, 2017.
[13] Marten Sigwart. Fakeload api. https://www.javadoc.io/doc/com.martensigwart/fakeload/. Retrieved August 12, 2017.
[14] Debian Webmaster. Details of package stress in stretch. https://packages.debian.org/stretch/stress. Retrieved August 12, 2017.