Scalar: Systematic scalability analysis with the Universal Scalability Law

Thomas Heyman
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
thomas.heyman@cs.kuleuven.be

Davy Preuveneers
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
davy.preuveneers@cs.kuleuven.be

Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
wouter.joosen@cs.kuleuven.be

***Abstract*—Analyzing the scalability and quality of service of large scale distributed systems requires a highly scalable benchmarking framework with built-in communication and synchronisation functionality, which are features that are lacking in current load generation tools. This paper documents Scalar, our distributed, extensible scalability analysis tool that can generate high request volumes using multiple communicating, coordinated nodes. We show how Scalar offers analytics capabilities that support the Universal Scalability Law. We illustrate Scalar on an electronic payment case study, and find that the framework supports complex work flows and is able to characterize and give predictive insights into the quality of service and relative capacity of the system under test in function of the user load.**

I. Introduction

Over the last decade, the scale of online systems has increased dramatically. Not only do we use online services more for everyday tasks, but the degree to which we depend on these services increases as well. This makes software qualities such as availability, scalability and performance essential for these systems. As the scale of these systems increases (both in planned number of users and complexity), assessing their actual capacity, performance, and future scalability potential becomes even harder: Generating user loads to simulate the scalability scenarios of a single web server for a local website is trivial compared to the complexity of simulating user loads for complex cloud enabled distributed deployments. The complexity increase is two dimensional, i.e., simulating ever more complex workflows and generating large enough loads to sufficiently stress the system under test.

Complex workflows are not only due to the user needing to fulfill more actions or follow a more involved business process, they often also depend on the collaboration of multiple (simulated) users. That, in turn, requires inter-user communication and synchronisation facilities in the load generation and benchmarking platform. Similarly, complex workflows might require out of band data processing and high volume data storage capacity. As client side computational overhead increases, care must be taken that the load generator itself does not become the bottleneck. In that respect, increasing workflow complexity and generating sufficient loads are not orthogonal problems—they reinforce each other.

This paper documents Scalar, a highly scalable distributed load generation and benchmarking platform that is developed specifically to handle these complex, large scale benchmarking problems. It supports inter- and intra-node communication and synchronization for aggregation of results, with built-in node monitoring to detect bottlenecks. Its analytics capabilities are backed by the Universal Scalability Law [1], [2] to enable systematic comparisons and trade-offs. Furthermore, Scalar features distributed statistics processing that take advantage of data locality, in order to enable it to easily scale to much larger load generation setups.

The paper is structured as follows. We give an overview of benchmarking and load generation systems in Section II. The problem statement is made explicit and illustrated on an e-payment case study in Section III. The tool architecture and design are documented in Section IV. We apply our solution to the case study and document our results in Section V. The results are discussed in Section VI. We conclude in Section VII.

II. Related work

There are a number of load testing frameworks in existence, ranging from load tests embedded in integrated development environments (such as Microsoft Visual Studio) to web testing frameworks with support for distribution (such as Apache JMeter). An overview of load testing tools can be found online [3]. We briefly overview four of these frameworks (Selenium, Gatling, JMeter and The Grinder), and identify their main strengths and weaknesses.

Selenium [4] is a browser automation framework. It comes in two flavours. Selenium IDE is a Firefox add-on that allows recording of interactions between a user and a website that can be played back later. Selenium WebDriver is a collection of language specific bindings that allow controlling a browser programmatically. Clearly, Selenium IDE is the closest thing to having an actual human behind a computer—it is easy to automate and intuitive, but does not scale well to supporting complex business flows with conditional execution, and does not allow easy communication and synchronisation between various instances. As Selenium IDE requires instantiating a browser, it has a large overhead, especially when thousands of concurrent users need to be simulated[1].

Selenium WebDriver does away with some of these disadvantages, at the cost of user friendliness. By embedding browser control in a host programming language, arbitrarily complex workflows can be supported. WebDriver nodes can also leverage the Selenium Server to automate the distribution

[1]Because Selenium leverages real browsers, it is the obvious choice when it comes to QA testing and ensuring that a web application works correctly in a specific browser version. That, however, is out of scope for this work.

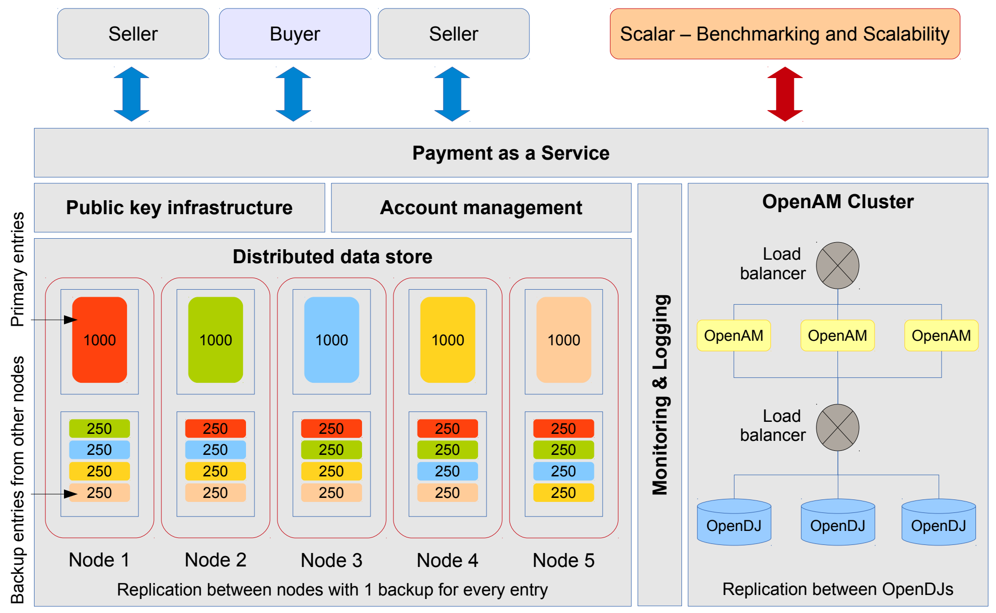

Figure 1. High-level overview of the MobiCent framework.

of load tests to a cluster of Selenium machines. However, Selenium is not built specifically for load testing, and does not come with built in scalability and quality of service analysis tools. There is also no out of the box support for communication and synchronisation between these nodes. As WebDriver nodes depend on instantiating browsers, this imparts a huge performance penalty and prevents easily scaling up to thousands of concurrent users. Finally, Selenium is focused on web applications, and it is not suited to automate load testing of arbitrary (i.e., non web) services.

Gatling [5] is an open source tool that focuses on load and stress testing. Scenarios can be encoded in either a domain specific language, or recorded via a proxy that intercepts browser traffic. It provides reports for load testing analysis which include a breakdown of active sessions, requests per second, and request residence time distribution. Gatling focuses mainly on testing HTTP(S) scenarios. It does not support clustering out of the box, but there is a workaround to aggregate output from multiple independent Gatling instances. However, these can neither synchronize nor communicate with each other.

JMeter [6] is one of the best known open source load testing tools. It is very flexible, and supports not only web applications, but also SOAP, FTP, various database protocols, SMTP(S), etc. It also supports clustering out of the box, by leveraging RMI. This limits JMeter clusters to the same subnet. There is also no inter machine communication facility, except for passing static data in configuration files. Although JMeter is fully extensible by means of plugins, there is no default support for scalability analysis (e.g., by means of applying the Universal Scalability Law).

The Grinder [7] is a Java load testing framework for HTTP web servers, SOAP and REST web services, and application servers. It offers a simple and minimal runtime, with flexible scripting in Jython. It is fairly straightforward to run distributed tests that leverage many load injector machines. As with JMeter, The Grinder has distributed agents that collate the data and send it back to the coordinator. The Grinder differs from JMeter in architectural design: its scripting based nature is a major difference with the component based structure of JMeter. Similarly to JMeter, however, The Grinder does not offer default built-in support for scalability analysis.

III. Illustration and Problem Statement

In order to illustrate the problem, consider the MobiCent framework, an online RESTful payment system that allows the creation and consumption of monetary transactions as a service. MobiCent allows business owners to create electronic coupons that can later be redeemed by customers, or sellers to create payment requests that can be sent to a buyer for completion at a later time. This decoupling of creating and consuming payment requests allows for more flexibility in applying the payment framework to different use cases. In order to achieve this, it is built on top of a public key infrastructure, coupled to an off-the-shelf user management and access control system[2]. However, in order to be successful in real-life deployments, its implementation must be scalable. Therefore, it uses a scalable distributed data store in the backend, which enables clustering of MobiCent servers—transactions are automatically partitioned and replicated among the different MobiCent instances.

A high-level graphical overview of MobiCent is shown in Figure 1. It depicts a logical high-level overview of a grid enabled payment architecture, the main core assets and how they are linked with one another. It exposes internal core assets through RESTful services for experimental purposes. The application offers a RESTful interface for 1) configuration management to customize the MobiCent deployment and the

[2]For user management and access control, MobiCent leverages OpenAM and the accompanying OpenDJ data store, http://forgerock.com/products/open-identity-stack/.

assets it provides, 2) user authentication to identify buyers and sellers using ForgeRock's OpenAM and OpenDJ identity and access management frameworks, 3) user provisioning for online bank accounts, 4) key-pair management for signing and verifying digital monetary transactions, 5) transaction management with support for distributed locking and various replication modes, and 6) performance monitoring.

The actual deployment of this architecture can differ in the number of nodes, as well as with regard to the replication and backup mechanism used for the in-memory data store and the OpenDJ stores. Regarding the latter, we did not differentiate between dedicated OpenDJs for (a) user stores and for (b) configuration stores. As the configuration stores also store and replicate live OpenAM sessions (to support failover if an OpenAM instance goes down), it will more likely create a bigger replication impact.

A detailed discussion on the operation of this framework is out of scope of this work. A high-level overview of the creation of a transaction is shown in Figure 2. A seller creates a payment request, adds an authentication token, adds his certificate and digitally signs the request. This request is then uploaded to a payment service which uses the token to verify that the seller is authenticated, and that the payment request is valid. A URL to the payment request is returned, which can then be used by the buyer to consume the payment request and complete the transaction. Completing a transaction mirrors the payment request creation, but instead creates an authenticated response to the original request. Additional information about this case study can be found at https://pong.cs.kuleuven.be/mobicent/.

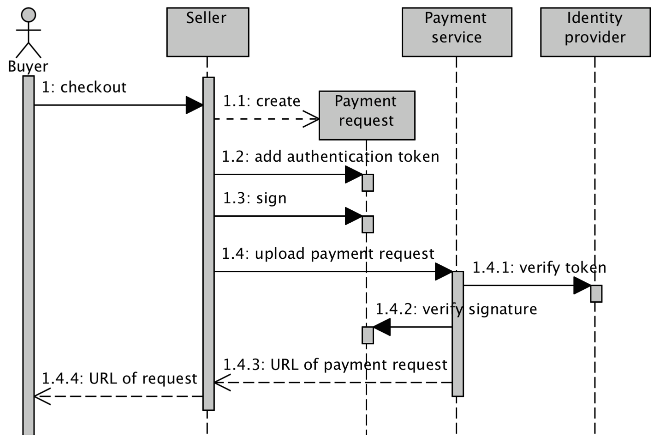

Figure 2. Creating a request.

Load testing and benchmarking the MobiCent framework is not a trivial endeavour for the following reasons.

1) MobiCent needs to support distinct usage scenarios (e.g., payment request creation and payment request consumption). Depending on intended use cases, a combination of different user types is required. Either way, we are interested in a statistical breakdown per request type, independent of the specific mix of both scenarios.
2) Both usage scenarios are not completely independent: Clearly, payment request consumption depends on payment request creation. In order to execute the consumption scenario, we need some sort of data storage so that a priori created requests can be retrieved. We also need a communication infrastructure, so that the creation and consumption can be coordinated.
3) Advanced coordination models require synchronisation in addition to data storage and communication.
4) As we are interested in benchmarking, care must be taken that the load generation itself is not the bottleneck. To facilitate this, we would need at least a warning mechanism when the load generator cannot handle the required load, a way to offload computation intensive tasks, and an auto benchmarking feature to find how far the load generator can scale on the underlying hardware.
5) The benchmarking should be fully automatable, so that experiments can be batch executed and results automatically gathered and processed. This improves repeatability, which in turn increases the confidence that the analysis results are representative.
6) The load generation process should be highly scalable itself, to accommodate benchmarking the largest distributed systems. This includes both horizontal scalability (i.e., deploying more instances in parallel), as well as vertical scalability (i.e., extensibility by means of plugins).
7) The scalability analysis should be scalable as well. Load tests of the envisioned distributed setups easily involve hundreds of thousands of requests per minute. The generated data volumes quickly outgrow the simple strategy of sending all data to a single processor for aggregation.

IV. Tool Architecture and Design

This section summarizes the design of Scalar, and highlights how it is able to support complex workflows and scale up to load test large deployments. More details, as well as a step-by-step tutorial on how to use Scalar, is provided online[3]. Scalar is a fully distributed system. It consists of multiple individual, collaborating Scalar instances. When deployed, Scalar instances discover each other, and a master is elected automatically. The master coordinates the start of an experiment (i.e., a scalability analysis), which consists of a number of individual runs (i.e., single load tests). Each run contributes one data point to the experiment. A run consists of a lower load warm-up phase, followed by a gradual ramp up to full load, the peak load phase during which statistics are collected, a ramp down phase, and finally another lower load cool down phase. Both the duration and the difference between the low and high load phases is configurable, any of these phases can be skipped by setting its duration to zero. When an experiment is complete, the master collates the results, quantifies the relative throughput of the system under test in function of user load by applying the Universal Scalability Law, and provides a statistical breakdown of request results and their residence times.

The Universal Scalability Law combines (a) the initial liniar scalability of a system under increasing load, (b) the cost of sharing resources, (c) the diminishing returns due to contention, and (d) the regative returns from incoherency into a model that defines the relative capacity $C(N)$:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)} \tag{1}$$

[3]See https://distrinet.cs.kuleuven.be/software/scalar/.

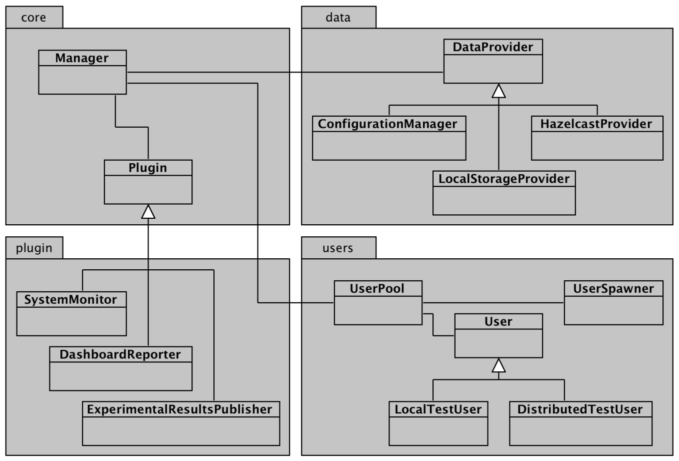

Figure 3. High-level design of Scalar. Details of the users, data and plugin packages are given in Section IV-A, Section IV-B and Section IV-C.

where N represents the scalability of the software system in terms of the number of concurrent users, α represents the contention penalty, and β defines the coherency penalty, with $0 \leq \alpha, \beta < 1$. To benchmark the scalability, the number of users N is incremented on a fixed configuration.

A high-level structural view on its design is provided in Figure 3. The core package implements the creation, scheduling and managing of simulated users. Representative user behaviour against which the system is to be tested, is encoded by creating one or more specific user and request types. That process is discussed in Section IV-A. Next, we document how we implement inter-user and machine communication and synchronisation to support complex workflows by means of data providers in Section IV-B. Finally, we show how we enable systematic scalability analyses with many Scalar nodes by means of plugins in Section IV-C.

A. Encoding workflows via users and requests

The abstract User class represents individual simulated users that follow a business flow which encodes the anticipated representative way in which the system will be used. All benchmarking results are relative towards the behaviour exhibited by the User objects. The typical life cycle of a User object is shown in Figure 4. Depending on the configuration, the Manager instructs the UserPool to create a number of User objects. The UserPool delegates User object creation to the UserSpawner, which uses reflection to load and instantiate the configured concrete user types at runtime.

Newly created user objects are scheduled for execution by adding them to a thread pool, which is responsible for periodically scheduling every User object for execution in its own thread. While user threads are scheduled with a configurable fixed wait in between two consecutive requests, they are started with small random delays. If the amount of users is large with respect to the inter-request wait time, the request arrivals are Poisson distributed. The business flow to be followed by individual user objects is encoded in the abstract goFetch method. By overriding this method, concrete User types can encode how to interact with the system to be benchmarked.

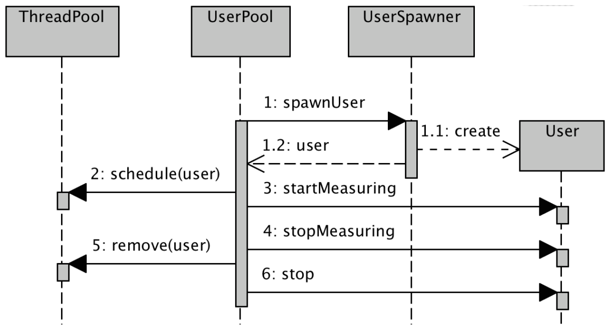

Figure 4. User life cycle.

User specific initialisation tasks can be handled by overriding the constructor. The User class also offers the option to override the onStop method to handle tasks that need to be done when a user object is destroyed. Finally, in order to make a distinction between requests generated during the warm-up phase, not all user actions are considered statistically relevant. The UserPool instructs user objects when to start and stop measuring by invoking the similarly named methods. Although collecting and processing requests is handled completely on an abstract user level, User subclasses are able to inspect whether the current requests are statistically relevant or not.

B. Supporting coordination with data providers

Inter-user communication is implemented by means of the blackboard architectural pattern: There is one central data repository, implemented by the DataProvider abstraction, which allows user objects to store and retrieve arbitrary objects. All communication between users is done via that repository. The interface of a DataProvider is similar to that of a map, and is based around a put(key, value) and get(key) operation. This abstraction allows for many interchangeable data provider implementations. Depending on the load testing requirements, a LocalStorageProvider can be used which leverages an underlying SynchronizedHashMap to store data per virtual machine, or a HazelCastProvider which leverages the HazelCast distributed storage system [8] that allows inter-machine communication. Adding a new data provider is as easy as extending the abstract DataProvider class, and specifying the new data provider implementation in the Scalar configuration file.

Synchronization is also built on top of the data provider abstraction. A data provider offers both lock(key) and unlock(key) operations, which allows synchronisation of both Scalar instances and user objects on specific key values. The LocalStorageProvider implements locks with standard Java ReentrantReadWriteLocks, while the HazelcastProvider leverages the underlying distributed Hazelcast locking mechanisms. Given that calls to the underlying HazelCast engine are inherently thread safe, the notion of distributed locks is sufficient for most synchronization requirements. However, other synchronization primitives such as spinlocks could equally be implemented by the data provider mechanism. As the overall Scalar functionality (including master election, instance discovery, experimental synchronization and results exchange) is built on top of this abstraction, fine tuning the Scalar cluster behavior can be achieved by selecting a correct underlying data provider. For instance, adapting Scalar to handle hard real time constraints could be achieved by creating in a new data provider that is able to offer real time guarantees on data storage, retrieval, and synchronisation.

Internal synchronisation and deadlock issues in user code is easily detected, as Scalar automatically detects when internal handling of requests takes too long: When the time between two user requests exceeds the configured delay by more than 5%, a warning is generated. That not only allows the experimenter to be warned of deadlocks in the test cases themselves, but also to detect bottlenecks in the load generation process. In the case where Scalar becomes too slow to generate sufficient load for the system under test, overall throughput can be optimized by refactoring user code (i.e., perform more processing asynchronously in helper threads or out of band in plugins), starting fewer user threads per Scalar instance, and increasing the Scalar cluster size.

C. Enabling large scale analyses with plugins

The overall functionality of Scalar can be modified and extended by means of plugins. A plugin is notified of different system events by means of several callback methods: When it is loaded and destroyed, and when the different load testing phases (i.e., warm-up, ramp up, peak load, ramp down, and cool down) take place. This allows plugins to perform platform wide initialisation tasks, such as populating the data provider with certain transactions to be executed, configuring the server under test, etc. Similarly, plugins can clean up the platform state in between different runs.

Plugins can also be used to inspect requests—every plugin receives a call-back for every executed request. This allows plugins to perform real-time request analysis and reporting. Clearly, care must be taken to process requests in real-time, as hundreds of requests can be generated per Scalar instance per second. For more computationally intensive processing, the plugin receives a call-back with a list of all requests generated during the previous run when that run is finished. Plugins can use the underlying data providers to store results.

The Scalar platform comes with three domain independent plugins built in which are crucial in enabling large scale analyses: The SystemMonitor, the DashboardReporter and the ExperimentalResultsPublisher. The first two plugins, the SystemMonitor and DashboardReporter, are essential in monitoring an experimental setup and gaining confidence in the correctness of the results. The SystemMonitor periodically gathers system resource usage, such as CPU and memory usage, as well as data sent and received per network interface. This allows identifying bottlenecks in the load generation process itself—for instance, when the network consumption data would indicate that the amount of generated traffic is close to the maximum network capacity. The DashboardReporter calculates simple statistics on request residence times and request results, and sends them to a web based dashboard for real-time visualisation. The dashboard allows experimenters to get a quick overview on how the load test is progressing, and whether results are not anomalous (e.g., when all requests have an error status because the system under test is not responding).

The third plugin, the ExperimentalResultsPublisher, handles distributed processing of request data and quantifies the scalability of the system under test in two dimensions. First, it calculates statistics per request type, and provides an overview of the distribution of request type residence times: it fits residence times to a specific distribution and calculates its parameters. That allows experimenters to calculate the residence time density function, which, in turn, provides answers to questions such as “How many requests were handled within 10ms?”. Second, the plugin computes the relative capacity of the system under test for various user loads, and fits the relative capacity data to the Universal Scalability Law. That allows experimenters to extrapolate how many users the system under test would be able to handle under different circumstances. It additionally allows pinpointing of the optimal load point, and provides a precise characterisation of the coherency and serial fraction parameters of that system, as per [9].

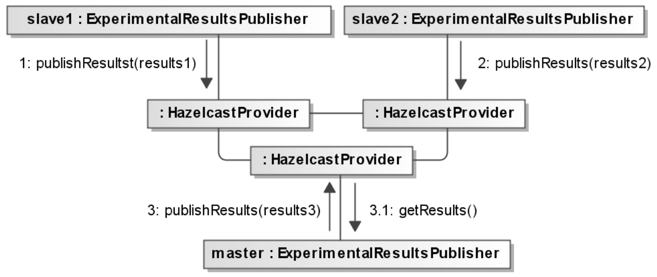
slave1 : ExperimentalResultsPublisher
slave2 : ExperimentalResultsPublisher
1: publishResultst(results1)
2: publishResults(results2)
: HazelcastProvider
: HazelcastProvider
: HazelcastProvider
3: publishResults(results3)
3.1: getResults()
master : ExperimentalResultsPublisher

Figure 5. Aggregating and publishing results.

The ExperimentalResultsPublisher is critical to achieve scalability of Scalar to analyze large, distributed setups, where sending all requests to a single point for analysis is not an option. It achieves the necessary scalability by means of distributed processing: Every Scalar instance is responsible to perform statistical processing on locally generated requests, i.e., calculate the mean and standard deviation of the local residence times, and provide total counts for all issued requests and their result status. Only these values are sent to the master Scalar instance. When all runs are completed and the experiment terminates, the ExperimentalResultsPublisher on the master Scalar instance aggregates the partial results statistics, and uses these to compute a global view on the system scalability and request residence time distribution. That reduces the problem of exchanging all request data to exchanging six floating point numbers and eight integers per Scalar instance, per request type. A graphical overview of result aggregation is shown in Figure 5.

V. Case study

To illustrate the working of the Scalar platform, we apply it to the MobiCent framework as documented in Section III. Instantiating Scalar for a specific problem domain is straightforward. First, one or more user types are created in Java that encode how a user is expected to interact with the system under test. Scalability results are relative to these user types. Second, one or more request types are created that are used by the user types to communicate with the system under test. Quality of service results are broken down per request type. Finally, extra functionality can be added to Scalar by means of plugins.

We begin by encoding the MobiCent use cases in a new MobiCentUser in Section V-A. How requests to MobiCent are encoded, is documented in Section V-B. We show how platform specific configuration and initialisation can be performed by means of plugins in Section V-C. Finally, we show how to operate the platform and example output in Section V-D. Code examples are provided, where applicable, for illustration purposes. Note that these are necessarily incomplete due to size considerations.

A. Creating a MobiCent user

In order to encode the MobiCent use cases, we introduce a MobiCentUser as a subclass of User. New user objects can be configured by overriding the constructor and leveraging the built-in data providers to read and optionally write values. Note that domain specific configuration options can simply be added to the traffic generator configuration file as key-value pairs, and retrieved via the data().getAsString(...) method, as shown below.

Listing 1. Creating a new User type.
```
package be.kuleuven.distrinet.trafficgenerator.users;

class MobicentUser extends User {
  MobiCentUser(DataProvider data, Plugin plugin)
      throws DataException {
    super(data, plugin);
    // Store and retrieve data via the data() DataProvider.
    String mobicent_url=data().getAsString("mobicent_url");
    // Create and execute requests.
    MobiCentAuthenticate r1=new MobiCentAuthenticate(this);
    // Note that User has support for generating and
    // (re−) using configurable user names and credentials
    // via the built−in username() and password() methods.
    String token = r1.doRequest(mobicent_url, username(),
          password());
    ...
  }
}
```

If multiple use cases have to be supported in parallel, then multiple user types can be created. The tool instantiates user objects based on its configuration. For instance, assume that we want to distinguish between transaction producers (i.e., product sellers) and consumers (i.e., users), then two types MobiCentUser and MobiCentSeller can be created. If we want to test the system behaviour with a mix of 5% sellers and 95% users, then the following configuration can be used.

```
user_implementations=
  MobiCentUser,MobiCentSeller
MobiCentUser=0.95
MobiCentSeller=0.05
```

If a seller wants to collaborate with other sellers or with a user to exchange transactions, then this is possible by means of the built-in data providers. Assume that transactions get assigned a strictly incrementing globally unique sequence number, for testing purposes. This implies that sellers need a synchronisation and locking feature to read and update the value of this counter, and users need to be able to receive transactions from sellers. This can be implemented as follows.

Listing 2. Synchronization and communication between users.
```
class MobiCentSeller extends User {
  public void publishTransaction() {
    data().lock("transaction_counter");
    int counter = data().getAsInt("transaction_counter");
    data().put("transaction_counter", counter+1);
    data().unlock("transaction_counter");
    data().put("transaction" + counter,
      new Transaction(this,counter));
  }
}

class MobiCentUser extends User {
  public void consumeTransaction() {
    for (String key : data().keys()) { // Find a transaction... }
    Transaction t = (Transaction) data().get(key);
    ...
  }
}
```

Clearly, this is a (rather naive) simplification: locking on a system wide value and generating transactions inline during load testing (which requires expensive cryptographic operations), risks creating a bottleneck in the traffic generation process itself. Additionally, the MobiCent service should be configured before the start of an experimental run, and its state reset after every run to make everything repeatable. However, as many runs can be scheduled automatically as part of an experiment, we need a way to automate this. That functionality can be handled by creating a plugin, as shown in Section V-C.

B. Creating requests

Requests are easily encoded by creating a new request type that extends the Request class. Requests are automatically aggregated and processed by Scalar—a breakdown is provided per request type, and per request result. Request results are indicated by means of the done(RequestResult) method; RequestResults include SUCCEEDED, ERROR, NO_RESULT , etc. The programmer can leverage the startTimer() and stopTimer() methods to finely measure the exact duration of an actual service request, and avoid interference from any pre or post processing.

Listing 3. Creating a new Request type.
```
package be.kuleuven.distrinet.trafficgenerator.requests;

public class MobiCentAuthenticate extends Request {
  public String doRequest(String url,
      String username, String password) {
    try {
      String dst= url +
        "/authentication/authenticate?username=" +
        URLEncoder.encode(username, "UTF−8") +
        "&password=" +
        URLEncoder.encode(password,"UTF−8");
      // Start the timer.
      startTimer();
      // Perform the request.
      String jsonResult = HttpUtil.doGet(dst);
      stopTimer(); // Stop the timer.
      // Perform result post processing.
      if (result.getCode() == 200)
        done(RequestResult.SUCCEEDED);
      ...
    }
  }
}
```

Of course, one user can generate multiple kinds of requests. For instance, MobiCent users create and consume transactions by means of MobiCentGetTransaction and MobiCentPutTransaction requests. Each MobiCentUser object also logs out of the system when it is destroyed, to avoid leaving sessions in an inconsistent state, by means of a MobiCentLogOut request. The code of these requests is omitted for brevity.

C. Platform configuration and initialisation via plugins

The main tasks of a plugin are platform initialization, tear down, user life cycle management, and offloading processing intensive operations to a point in time where the load testing results can not be biased. In the case of generating transactions, for instance, we want to distinguish between preparing a transaction (i.e., obtaining a sequence number and signing it) and registering a transaction (i.e., sending it to the MobiCent service for validation). The latter, load testing the service, is what we are interested in, and care must be taken that the former does not bias the results. Therefore, preparing transactions can be performed by a plugin right before each experimental run commences. The same holds for configuring the MobiCent service and deleting registered transactions after each run. As with user and data provider implementations, plugins can be loaded by specifying them in the Scalar configuration file.

Listing 4. Creating a plugin.

```
public class MobiCentPlugin extends Plugin {
  @Override
  public void onInitialization() {
    // Read configuration from data(), send to server.
    // Note that this should only be done by one traffic
    // generator instance, letting the master do it is
    // an easy way out:
    if (data().getAsBoolean(
            Option.LOCAL_MASTER.toString())) {
      // Retrieve configuration options.
      boolean senderAuthentication =
        data().getAsBoolean("sender_authentication");
      ...
      // Send configuration options to the server...
    }
  }

  @Override
  public void onStartUp() {
    // Prepare and store transactions.
  }

  @Override
  public void onStop(ArrayList<Request> allRequests) {
    // Reset the service configuration.
  }
  ...
}
```

D. Performing the experiment

To illustrate, consider the following experiment. A Scalar configuration file is created by the experimenter for MobiCentUsers to generate a transaction every 500ms for a period of one minute per run, and that for various increasing loads. A number of Scalar instances are started, which perform auto-discovery and master election. The master then coordinates the individual experimental runs, and aggregates the experimental results. An overview of the output generated by Scalar for a load test of transaction creation and consumption is shown in Table I; the output fitted to the Universal Scalability Law is depicted in Figure 6.

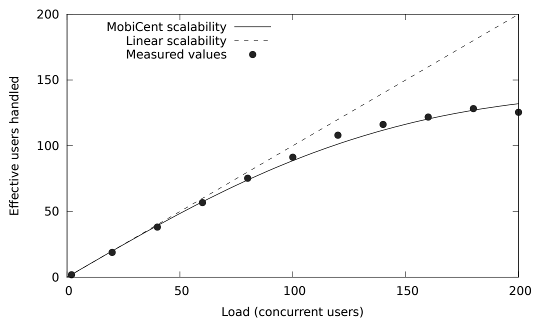

Figure 6. Scalability graph of the MobiCent framework.

The results give an indication of how many concurrent users that match the specified simulation could be handled by that specific MobiCent instantiation, i.e., with the given data replication strategy in the backend, load balancing strategy, and so on. The output of the dashboard during a load test of the authentication step of the MobiCent flow is shown in Figure 7. It contains an indication of the load on the Scalar system in the top left corner (as calculated by taking the average CPU usage of all Scalar instances), an overview of request residence times in the candlestick chart on the bottom left, the USL model of the service with an indication of the current load level on the bottom right, and a prediction of how many requests are successfully handled within an example quality of service policy of 5ms on the top right.

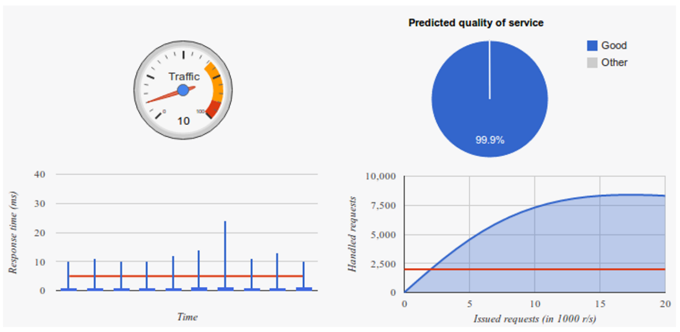

Figure 7. Monitoring the scalability analysis results in real time with the DashboardReporter plugin.

VI. Discussion

The coding effort required to create new user and request types in Scalar is low. For instance, a trivial user class that performs simple HTTP HEAD requests to a web server, is 30 lines of code (including whitespace), while the HTTP HEAD request class is 43 lines of code.

Users	Min (ms)	Max (ms)	Mean (ms)	Std. dev.	Shape	Scale	Requests
…	…	…	…	…	…	…	…
100	5.0	224.0	18.79	14.775	1.617	11.617	6798
120	5.0	351.0	21.048	20.902	1.014	20.756	8659
140	5.0	218.0	26.277	22.084	1.416	18.56	11033
…	…	…	…	…	…	…	…
200	3.0	618.0	43.288	59.41	0.531	81.537	17446

Table I. EXAMPLE PERFORMANCE RESULTS GENERATED BY SCALAR.

When compared to frameworks such as JMeter[4], it is clear that the data provider abstraction of Scalar allows for more flexible communication and synchronisation. Scalar inherits the scalability of the underlying Hazelcast system, which is explicitly designed to scale up to clusters of hundreds of nodes[5]. However, in specialized contexts (e.g., a real-time or embedded domain), it is fairly straightforward to plug in a different communication and synchronisation layer, as the dependency on Hazelcast is not hard coded. Similarly, the distributed statistics aggregation means that longer, high volume experiments involving many Scalar nodes can be achieved.

During the presentation of the case study in Section V, we tacitly assumed that it is possible to configure the MobiCent service remotely. Clearly, this is not always so. In order to streamline the automation of the scalability analysis, we have added a simple REST interface that allows changing the most common configuration options at runtime. Of course, adding operations to a service is not always possible, nor desired. In that case, however, simple management tools could equally be automated—for instance, a simple JMX component could be triggered by the traffic generator framework to provide the same effect. This is the topic of future work.

Load testing is a very subtle process. Even though Scalar immensely facilitates the scalability and quality of service analysis of a system, care must be taken to avoid pitfalls. The most common pitfall is that the bottleneck is not in the system under test, and that the load testing results are skewed by the load generation process itself. In order to avoid this, the following guidelines can be used. First, make use of the built-in LocalTestUsers to verify that the underlying traffic generation systems can easily handle the envisioned amount of user threads without significant slowdown. Second, perform the same test with the DistributedTestUsers, if node-to-node communication is used, to test that the network is able to comfortably handle the traffic generation cluster size. Third, enable the SystemMonitor to monitor resource usage and identify potential bottlenecks.

To ensure that the load testing results adequately characterize the system under test, it is first necessary to obtain a rough estimate of the optimal load point p^*. This can be achieved by performing initial, rough experiments and observing when the relative capacity starts to stagnate in function of increasing load. Once this point is found, an experiment with preferably at least 10 runs distributed over the interval $[1..1.5p^*]$ should be performed. That ensures that the relative capacity curve adequately characterizes the maximum service throughput. Also, be wary of interference from other system processes (it would not be the first time that an automated *apt-get update* skews some of our data points). This can be partly mitigated by observing the results in real time via the dashboard to spot anomalies.

[4]How JMeter could be deployed in a similar distributed context is summarized in http://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.pdf.

[5]E.g., http://cloud.dzone.com/articles/running-hazelcast-100-node.

VII. CONCLUSION

We have presented Scalar, a distributed platform for large-scale load testing and quality of service analysis. The platform is developed specifically to support complex workflows that involve both intra- and inter-machine communication and synchronisation. It also supports scaling to large deployments by means of distributed results processing. In order to achieve that, it is built upon the Hazelcast distributed in-memory data grid. The platform functionality is extensible by means of custom user types, as well as plugins. Built-in functionality includes monitoring of the underlying load generating platform, support for data aggregation and analysis by means of the Universal Scalability Law, and real-time visualisation of results via a web based dashboard.

Scalar has already been applied successfully to a number of in-house projects, as well as commercial systems. We conclude that it is capable of characterizing both the scalability and quality of service of complex, distributed services. Future work involves automating the instantiation of Scalar for very large cloud-based deployments. That would allow us to eventually achieve load testing as a service.

ACKNOWLEDGMENT

This research is partially funded by the Research Fund KU Leuven.

REFERENCES

[1] N. J. Gunther, “A simple capacity model of massively parallel transaction systems,” in *CMG-CONFERENCE-*. COMPSCER MEASUREMENT GROUP INC, 1993, pp. 1035–1035.

[2] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of queueing theory*. John Wiley & Sons, 2013.

[3] N. J. Gunther, “How to Quantify Scalability,” http://www.perfdynamics.com/Manifesto/USLscalability.html, online; accessed 13-May-2014.

[4] SeleniumHQ, “Selenium - browser automation,” http://docs.seleniumhq.org/, online; accessed 6-March-2014.

[5] Gatling, “Gatling Stress Tool,” http://gatling-tool.org/, online; accessed 6-March-2014.

[6] The Apache Software Foundation, “Apache JMeter,” http://jmeter.apache.org/, online; accessed 17-February-2014.

[7] Philip Aston, “The Grinder,” http://grinder.sourceforge.net/, online; accessed 6-March-2014.

[8] Hazelcast, Inc., “The Hazelcast Open Source In-Memory Data Grid,” http://www.hazelcast.org/, online; accessed 6-March-2014.

[9] N. J. Gunther, *Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services.* Springer, 2007.