Scalar: A distributed scalability analysis framework

Thomas Heyman, Davy Preuveneers, and Wouter Joosen

iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
`first.last@cs.kuleuven.be`

Abstract. Analyzing the scalability and quality of service of large scale distributed systems, such as cloud based services, requires a systematic benchmarking framework that is at least as scalable to sufficiently stress the system under test. This paper summarizes Scalar, our distributed, extensible load testing tool that can generate high request volumes using multiple coordinated nodes. It has support for communication and synchronization between user threads, and built-in node monitoring to detect resource bottlenecks in the benchmark framework deployment itself. Furthermore, it offers highly scalable results analysis that exploits data locality and characterizes the overall system scalability in terms of the Universal Scalability Law.

1 Introduction and problem statement

Over the last decade, both the scale of online systems and the degree to which we depend on them has increased tremendously. This makes software qualities such as availability, scalability and performance essential. However, as the scale of a system increases in number of users and complexity, assessing its actual capacity and future scalability potential becomes even harder. The problem is twofold: We need to simulate ever more complex work flows while generating large enough loads to sufficiently stress the system under test.

Workflows become more complex due to the user fulfilling more actions or following more involved business processes. They often also depend on the collaboration of multiple users, which requires inter-user communication and synchronisation in the load generation and benchmarking platform. Similarly, complex workflows might require out of band data processing and a high volume data storage capacity. As the computational overhead increases, care must be taken that the load generator itself does not become the bottleneck. This makes increasing workflow complexity and generating sufficient loads a compound problem.

To solve the problem, the ideal scalability analysis tool would realise the following requirements. First, it needs to explicitly *support multiple concurrent usage scenarios*, and provide statistical breakdowns per scenario. Some distinct usage scenarios are not independent, and users that execute one scenario depend on the actions performed by users in another scenario. Therefore, second, the tool should explicitly *support inter user communication and data exchange*. Third, as the complexity of interdependent usage scenarios and the number of simulated users increase, the tools should also *support synchronisation.*

Scalability and performance are two crucial qualities for our ideal scalability analysis tool. Clearly, in order to analyze large scale systems, scalability analysis tools should be highly scalable themselves. This includes both horizontal scalability (i.e., deploying more instances in parallel), as well as vertical scalability (i.e., extensibility by means of plug-ins). As load tests of the envisioned distributed setups easily involve hundreds of thousands of requests per minute, tools should support intelligent results processing that takes data locality into account. When scaling up, care must be taken that the load generation itself is performant enough to not become the bottleneck. To facilitate this, we would need at least a warning mechanism when the tool cannot handle the required load, a way to offload computationally intensive tasks, and a way to find how far the tool can scale on the underlying hardware.

Many load testing tools exist, ranging from load tests embedded in integrated development environments (such as Microsoft Visual Studio) to web testing frameworks with support for distribution (such as The Grinder [1] and Apache JMeter [2]). However, many are lacking inter-user communication and synchronisation facilities, built-in analytics and bottleneck detection, or both. For instance, JMeter has no inter machine communication facility, except for passing static data in configuration files. And although it is fully extensible by means of plug-ins, there is no default support for scalability analysis (e.g., by means of applying the Universal Scalability Law [3]). Similarly, while The Grinder has distributed agents that collate the data and send it back to the coordinator, it does not offer default built-in support for scalability analysis. In the next section, we document Scalar, a highly scalable distributed load generation and benchmarking platform that is developed specifically to support these features.

2 Scalar architecture

Scalar (`https://distrinet.cs.kuleuven.be/software/scalar/`) is a fully distributed system implemented in Java, and consists of multiple individual, collaborating Scalar instances. Scalar instances automatically discover others, and perform master election. 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). 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. The master collates the results and publishes a scalability report consisting of a quantification of the relative throughput of the system under test in function of user load, as characterized by the Universal Scalability Law, and a statistical breakdown of request residence times and their results.

Representative user behaviour against which the system is to be tested, is encoded in one or more specific user and request types. The abstract User class represents individual simulated users that follow a business flow which encodes the anticipated way in which the system will be used. All scalability analysis results are relative towards that behaviour. 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. The interface of a DataProvider is similar to that of a map. This abstraction allows for many flexible data provider implementations to be used interchangeably. The default data provider, HazelCastProvider, leverages the underlying HazelCast distributed in-memory database [4], which allows inter-machine communication.

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 HazelcastProvider leverages the underlying distributed Hazelcast locking mechanisms. 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 implementation.

The overall functionality of the Scalar platform can be modified and extended by means of plug-ins. A plug-in is notified of different system events via 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 plug-ins 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, plug-ins can clean up the platform state in between different runs. Plug-ins can also be used to inspect requests—every plug-in receives a call-back for every request that has been executed. This allows plug-ins to perform real-time request analysis and reporting. Plug-ins can use the underlying data providers to store results.

Scalar comes with a number of domain independent plug-ins, such as monitoring the underlying platform resources and visualising results in real time on a web-based dashboard. The most important plug-in for large scale analyses is the ExperimentalResultsPublisher, which 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. That allows experimenters to calculate the residence time density function, which provides answers to questions such as "How many requests were handled within 10ms?". Second, the plug-in 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 [3]. Scalar exploits data locality by making every node responsible for calculating aggregate statistics on raw data, and only exchanging these aggregate values with the master node.

3 Discussion and conclusion

We have presented Scalar, a distributed platform for scalability analysis of large distributed systems. The platform is developed specifically to support complex workflows that involve both intra- and inter-machine communication and synchronisation, and is fully extensible by means of plug-ins. Built-in functionality includes monitoring of the underlying load generating platform, support for data aggregation and analysis by means of the Universal Scalability Law, real-time visualisation via a web based dashboard, and time synchronisation over NTP.

Scalar inherits the scalability of the underlying DataProvider system. In the case of the HazelcastProvider, the underlying system is explicitly designed to scale up to clusters of hundreds of nodes. 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 enables longer, high volume experiments involving many Scalar instances.

Scalability analysis is rife with pitfalls. The most common one is that the bottleneck is not the system under test, but the load generation process itself. In order to avoid this, Scalar comes with a number of built-in protection features. First, the tool contains various domain independent test user implementations that can be used to perform a scalability analysis of a Scalar deployment itself, to detect problems early. Second, Scalar will automatically generate warnings when scheduled requests exceed the inter-request waiting time (i.e., the 'think time') by more than 5%. Experience shows that that is a good indicator for detecting bottlenecks internal to the load generation process. Third, the tool comes with built-in resource monitoring of the underlying platform.

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 achieve scalability analysis as a service.

Acknowledgment

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

References

1. Philip Aston: The Grinder. `http://htmlunit.sourceforge.net/` Online; accessed 6-March-2014.
2. The Apache Software Foundation: Apache JMeter. `http://jmeter.apache.org/` Online; accessed 17-February-2014.
3. Gunther, N.J.: Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services. Springer (2007)
4. Hazelcast, Inc.: The Hazelcast Open Source In-Memory Data Grid. `http://www.hazelcast.org/` Online; accessed 6-March-2014.