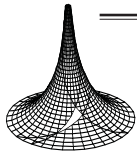


19/1 (2021), 17–33
DOI: 10.5485/TMCS.2021.0518
tmcs@science.unideb.hu
http://tmcs.math.unideb.hu

Teaching
Mathematics and
Computer Science

Teaching performance testing

Gábor Árpád Németh and Péter Sótér

Abstract. Performance testing plays a vital role in the verification of large scale software systems. It is used for testing the speed, responsiveness, capacity and stability of the investigated system. However, despite the significance of this topic, the effort invested in teaching performance testing in Computer Science is insufficient.

The current paper shows, how the fundamentals of performance testing can be demonstrated to students both from a theoretical and a practical viewpoint through step-by-step practical examples that are used in the industry. It is also discussed how a basic toolchain can be set up for performance tests using only free tools. With the presented examples, the reader will be able to take first steps in the performance testing area.

Key words and phrases: performance testing, load tests, load generation, teaching.

MSC Subject Classification: 68M15.

Introduction

Large scale software systems are designed to serve thousands or even millions of parallel requests. Performance testing is applied to verify if these requirements are fulfilled: numerous transactions are generated for the SUT (System Under Test) in order to check how it performs in terms of responsiveness and stability. This topic has been extensively studied in the last few decades, see for example (Jiang & Hassan, 2015) that overviews around 200 papers on this topic. More recent papers investigate different performance testing tools (Abbas, Sultan, &

Bhatti, 2017) or related hot topics, like using reinforcement learning to find performance bottlenecks (Koo, Saumya, Kulkarni, & Bagchi, 2019; Ahmad, Ashraf, Truscan, & Porres, 2019).

Problems related to system performance arise periodically in various fields, underlining the importance of performance testing. The authors remember the time as students in the 2000s when the software that was used for exam and course management crashed regularly when the period for choosing courses was opened. But to be fair, similar problems occur regularly in much bigger international software companies, although typically at a larger scale. Web shops have performance issues (crashes, the wrong handling of inventory items, limitations in the number of possible users, etc) during a Black Friday campaign (Wu, Patil, & Gunaseelan, 2018). The overload of collaboration and visual conference applications during home office in the time of COVID-19 and the wrong scheduling of patches that cause performance drops are also common examples.

Despite the importance of performance testing, very little effort has been invested in the teaching of this topic at universities, at least in Hungary. As an answer to this situation, a new course called "Modeling and testing" has been launched in the ELTE Computer Science MSc program which - besides model-based testing (Németh, 2020) also deals with performance testing. In the current paper, some of the main cornerstones of the material related to the latter topic is shown. The working principle of performance testing tools is illustrated with a small-scale example from the telecommunication industry and it is also discussed how an actual performance testing environment can be set up with freely available tools for the website testing domain.

Performance testing

Performance testing types

Performance testing is a general term that consists of different types of testing. Note that there are some inconsistencies in the terminologies in the documentation of corresponding testing tools, teaching materials (ISTQB, 2018) and sometimes even in the related research papers (Jiang & Hassan, 2015). However, the following classification can be used as a good starting point to understand the different motivations behind performance testing as similar goals are presented both in theory and in the industry (see Figure 1):

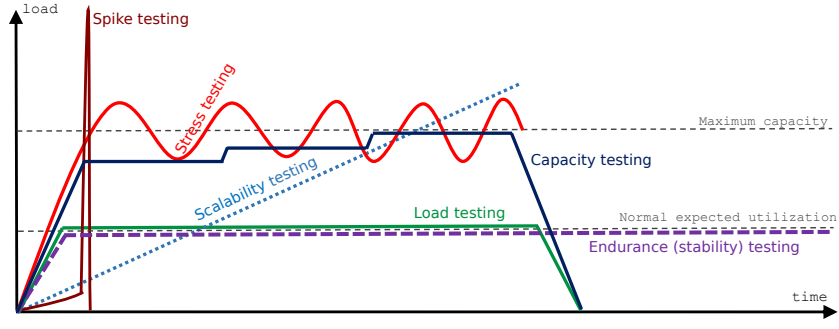


Figure 1. Performance testing types

- **Load testing**: Used to observe the behaviour of the SUT under an expected load to check its sustainability.
- **Endurance or stability testing**[1]: It investigates the stability of the system over a large predefined time frame. It is used to detect memory leaks, thread problems and any other types of problems that may threaten stability.
- **Capacity testing**[2]: It verifies if the SUT can manage the amount of workload that it was designed for. If not known in advance, it benchmarks the number of users or transactions that the current system version is able to handle, which can be used as a *baseline* for testing updated system versions.
- **Scalability testing**: It investigates how the SUT is capable of scaling up-/out/down considering resources (like CPU, memory or network) to find possible bottlenecks of the system (related to both current or future efficiency requirements). Two different approaches are present when investigating scalability: (1) Increase the load over a period of time gradually to monitor the amount of different resources used, (2) Scaling up/out the resources of the tested system with the same level of load.
- **Stress testing**: It is performed around the maximum designed capacity of the SUT to investigate how the system works near to this maximum.
- **Spike testing**: Focuses on the functioning of the system in case of sudden bursts of requests, where the load consumedly exceeds the maximum designed capacity. It checks whether the SUT crashes, terminates gracefully or just dismisses/delays the processing in this case.

[1]Sometimes also referred to as soak testing.
[2]Sometimes also referred to as volume or flood testing.

Working principle of performance testing tools

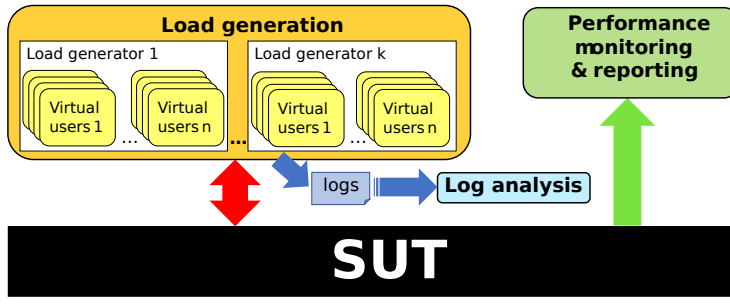


Figure 2. Performance testing

The working principle of performance testing tools can be seen in Figure 2: to be able to investigate the SUT (System Under Test) under various input conditions (such as the number of concurrent users, the frequency and distribution of different types of requests, etc.) a *workload* is created by different machines called *load generators*. This workload is done by creating *virtual users* to simulate or emulate the behaviors of *actors* (users or other machines that use the services of the SUT). Note that these virtual users are not necessarily just clones of each other, they are also able to act independently depending on their *user profile*. For example if a webshop is tested, one subset of these users will just be browsing, another subset will be selecting products for their cart, yet another subset will be paying, while an admin user may be checking some parameters of the related database, etc. During the execution of the workload to the SUT, some selected performance metrics are *monitored*. These monitored parameters can be connected to hardware utilization (such as CPU/memory/disk/network utilization) or related to the characteristics of the tested system (such as response time, throughput rate, rate of successfully handled requests, number of active sessions, etc) or can be connected to user's experience (such as how long it takes to finish the requested transaction). Performance monitoring may also give an alert in case of suspicious or lower performance scenarios. At the end, a *report* is created based on the observed metrics. After test execution, the collected logs may also be converted for further analysis.

There are different approaches that describe the behaviour of the virtual users used in load generation scaling from the simplest solution to more complex scenarios:

(1) **Packet generators:** This approach is the simplest. Some fundamental parameters of the generated packages can be modified, but the same setting is used for the entire load. For example, Netstress[3] is a free packet generator tool.

(2) **Traffic playback tools:** This approach plays back a – previously recorded or manually edited – call flow many times to generate the desired workload. For example, Apache JMeter[4] is a free traffic playback tool.

(3) **Model-based generators:** These generators use formal models to *emulate* virtual users; besides the normal call flow, alternate flows and exception flows are also investigated. Various models can be used; EFSMs (Extended Finite State Machines) (Németh, 2016), Markov chains (Barros et al., 2007), Petri nets (Youness, El-Kilani, & El-Wahed, 2008) and PTAs (Probabilistic Timed Automata) (Abbors, Ahmad, Truscan, & Porres, 2013) are the most common options. In (Erős & Csöndes, 2010) a TCFMM (Timed Communicating Finite Multistate Machine) model is presented that is an extension of the EFSM model with tokens and delays on transitions.

Examples for different load generation approaches: In the following, the three different approaches for load generation are shown through an example used in the telecommunication industry.

The SIP (Session Initiation Protocol) (*RFC 3261: SIP: Session Initiation Protocol*, 2002) is a signalling protocol used for establishing, modifying, and terminating real-time multimedia sessions such as internet telephony calls. SIP is a text-based protocol with a syntax similar to that of HTTP (HyperText Transfer Protocol). It operates with request and response message pairs. The type of the request sent by a User Agent Client (UAC) participant (that can be for example an IP (Internet Protocol) phone) is called *method*. A SIP server participant answers to this request with a given *response code*. Each UAC needs to register to a SIP server to ensure its location to be known in order to receive an incoming call later. In the following, the functionalities of the UAC during the registration process of the SIP over the TCP (Transmission Control Protocol) transport layer will be discussed in order to simulate this participant for load generation. The other participant of the communication, the SIP server will be the SUT in our example.

[3]Netstress. Network benchmarking tool. `http://nutsaboutnets.com/archives/netstress`
[4]Apache JMeter. Performance testing tool. `https://jmeter.apache.org`

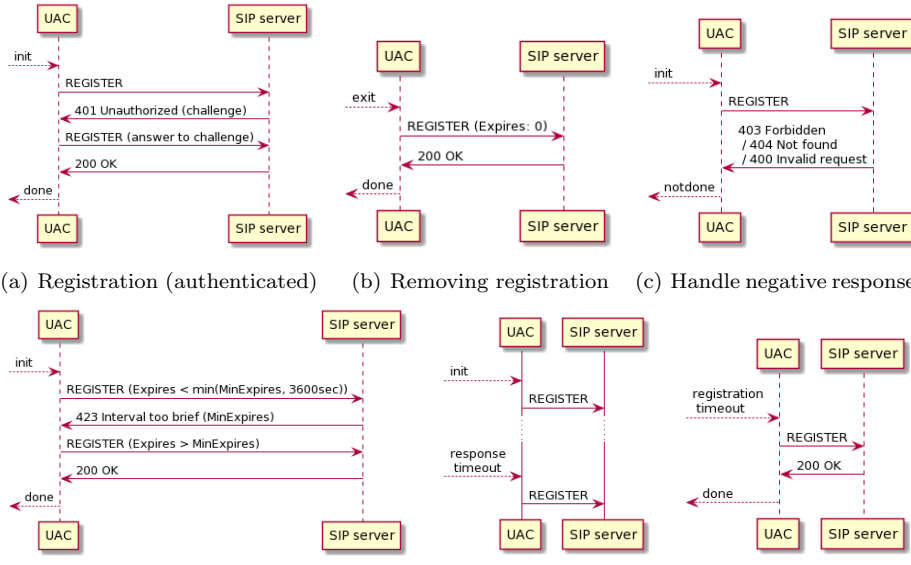
Note that this example has been selected due to the following reasons: (1) the related standards (*RFC 3261: SIP: Session Initiation Protocol*, 2002; *RFC 3665: Session Initiation Protocol (SIP) Basic Call Flow Examples*, 2003) are freely available and relatively easy to understand for everyone and (2) telecommunication is a perfect domain to show the *convergence of functional and performance testing*. The reason for the latter is that there is no sense to talk about whether a server fulfills all of the required functionalities if it is not able to handle the given number of users parallel, and vice versa: the performance of the server should be investigated with a given *functionality mix*.

(1) **Packet generators:** The load generation can be described with simple packets which imitate only the main structure of the registration process. For example the *REGISTER* message is sent multiple times with a preset encrypted user information regardless of the existence of any challenge it should respond to (Section 2.1 of RFC 3665). Due to this preset property of each package, one can only observe performance parameters, but is unable to check functionalities, such as authentication (in this case the appropriate authentication header should be determined previously by functional testing or the authentication should be turned off for the SUT side).

(2) **Traffic playback tools:** The load generation can be described with different call flows that implement the various functionalities of the UAC registration process – see Figure 3. The network communication between the UAC (tester) and the SIP server (SUT) participant is denoted with solid lines and the event that occurs within the UAC participant (i.e a message that is received from or sent to its operation system) is shown with dotted lines.

- Figure 3(a) shows the call flow of successful registration with authentication (Section 2.1 of RFC 3665): The UAC sends a *REGISTER* request to the SIP server, which provides a challenge in his *401 Unauthorized* response. The UAC responds with a *REGISTER* message that contains the encrypted user information according to the challenge. If the validation was successful, the SIP server sends a *200 OK* response.

- Figure 3(b) shows how the cancellation of registration works (Section 10.2.2 of RFC 3261 and Section 2.4 of RFC 3665): The UAC sends a *REGISTER* request to the SIP server with a 0 value in the *Expires* header. Note that all registrations expire after their time defined in *Expires* header has elapsed.



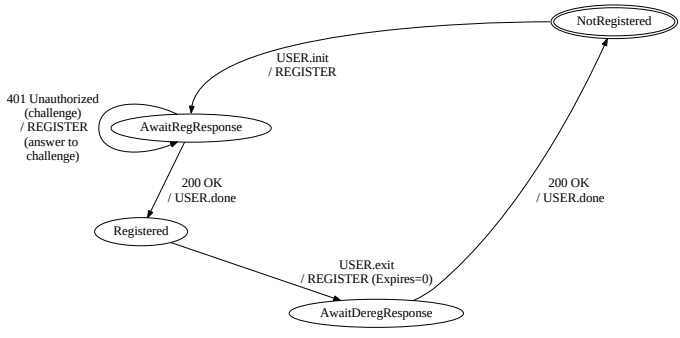
(a) Registration (authenticated) (b) Removing registration (c) Handle negative responses

(d) Interval too brief (e) Silent discard (f) Re-registration

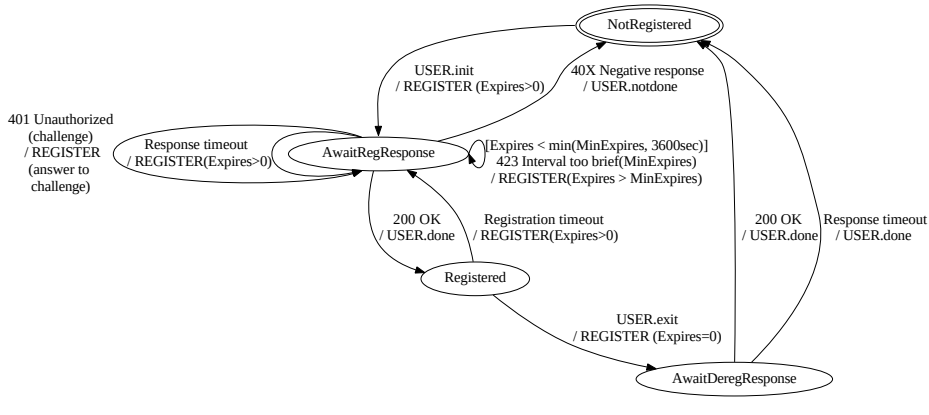
Figure 3. Call flows for the registration process of SIP UAC

- Figure 3(c) deals with the handling of negative responses for registration requests (Section 10.3 / $4^{th} - 6^{th}$ points of RFC 3261): Receiving a *403 Forbidden*, a *404 Not found* or a *400 Invalid response* message for the *REGISTER* request indicates that the registration was unsuccessful.
- Figure 3(d) deals with the case when a too brief interval is chosen for registration (Section 10.3 / 7^{th} point of RFC 3261): If the interval given in the *Expires* header of *REGISTER* request is too short (i.e. it is less than one hour and less than the minimum value defined by the SIP server), then the SIP server responds with a *423 Interval too brief* message with a requested minimum value defined in the *Min-Expires* header. It triggers the UAC to send another *REGISTER* request with an appropriate value in the *Expires* header.
- Figure 3(e) shows silent discard (Section 10.2.7 / 7^{th} point of RFC 3261): If no response has been received for the *REGISTER* request within a predefined time, a *Response timeout* event is received by the operation system of the UAC that triggers the UAC to resend the *REGISTER* request to the SIP server after waiting some reasonable time.

- Figure 3(f) describes re-registration (Sections 10.2.1.1 and 10.2.4 of RFC 3261): After a predefined time has elapsed since the registration on the server, an internal timer of the UAC indicates with a *Registration timeout* message that the registration will expire and the UAC should send a *REGISTER* request to the SIP server again to keep registration alive.



(a) EFSM for registration with authentication and for cancellation of registration



(b) EFSM for full set of functionalities

Figure 4. EFSMs for the registration process of SIP UAC

(3) **EFSM model-based generators:** Figure 4 shows the EFSMs[5] of the registration process of the SIP UAC that can be made by the combinations of

[5]Note that the strict formalism of EFSM models and how the desired output message for a given input can be checked is described in more detail in our previous article (Németh, 2020).

the call flows presented previously in Figure 3. The labels in each transition between states denote the input and output messages written as *input/output* derived from the request and response messages of the call flows. The parameters of the request or response messages will be the variables of the EFSM denoted in round brackets. The guarding conditions on variable values – that may add additional triggering conditions for each transition besides their input and start state – are denoted in square brackets.

In the following, it will be discussed step-by-step, how the call flows presented in Figure 3(a) and 3(b) can be mapped into the simple EFSM presented in Figure 4(a):

- Successful registration with authentication: The message pair, where the *init* event of the UAC triggers the sending of the *REGISTER* request to the SIP server is mapped into *USER.init / REGISTER* transition from state *NotRegistered* to state *AwaitRegResponse* in the EFSM. Similarly, the *401 Unauthorized (challenge)* response from the SIP server that triggers the UAC to send a *REGISTER (answer to challenge)* to the SIP server is mapped into the *401 Unauthorized (challenge) / REGISTER (answer to challenge)* loop transition of the state *AwaitRegResponse*. At the end, the *200 OK* answer received from the SIP server that triggers the *done* event on the UAC will be mapped into the *200 OK / USER.done* transition originating from state *AwaitRegResponse* and leading to state *Registered*.

- Cancellation of registration: The *init* event of the UAC that triggers the sending of the *REGISTER (Expires: 0)* request is mapped into *USER.init / REGISTER (Expires=0)* transition from state *Registered* to state *AwaitDeRegResponse*. When the *200 OK* response is received from the SIP server, the UAC goes to initial state *AwaitDeRegResponse* and throws a *USER.done* event.

The mapping can be done similarly for other call flows by paying attention to how these call flows can be merged using appropriate start and end states for each transition – see Figure 4(b) for the full set of functionalities derived from call flows of Figure 3(a)–3(f). Note that a similar EFSM can be constructed for the other side of the communication (SIP server) as well, and the EFSMs of the two participants would communicate with each other

with message exchanges[6], but now we focus on the emulation of virtual users for the UAC side to be able to generate appropriate load for the SUT.

Although the resulting EFSM of Figure 4(b) is a very compact model, it contains all the functionalities of Figure 3. It is not required to know the entire call flow in advance to be able to test the SUT as in traffic playback generation. The message received at each step of the communication can be used as a triggering condition to determine, which step should be taken next in the EFSM. Thus, the test engineer is able to create a *traffic mix* to test different functionalities and balance between functional and performance testing goals.

A toolchain for performance testing

Apache JMeter

For the course, Apache JMeter[4] was selected for hands-on experience of performance testing, because it is a well documented, easy-to-use, free and open source performance testing tool with many functionalities. JMeter uses the traffic playback approach for load generation.

To illustrate the performance testing of websites with JMeter, two simple load generation exercises are briefly discussed here.

I. **Recording HTTP traffic and playback with 10 parallel threads:**

1) Before the actual test design is started, the web browser must be configured to allow JMeter to observe its traffic by adding a JMeter certificate and by selecting manual proxy configuration.

2) After that, one is able to start the recording of browsing a webpage with the *HTTP(s) script recorder* of JMeter. Loading the pages will be slow due to the injection of JMeter into the communication.

3) Set the parameters of the *Thread Group (TG)* that will act as a group of virtual users in load generation: *The number of Threads (users)* is set to 10 to create 10 parallel playbacks of the recorded traffic. The *Ramp up period* that defines delays between starting parallel users is set to 1 sec. The *Loop count* which defines how many times the test will be repeated is set to 20 for example.

[6]These models are known as CEFSMs (Communicating Extended Finite State Machines).

4) Add *Aggregate report*, *Summary report* and *Graph Results* statistics to *TG*.
5) Save and validate the TestPlan.
6) Start the execution of test and observe statistics – see Figure 5 for results.

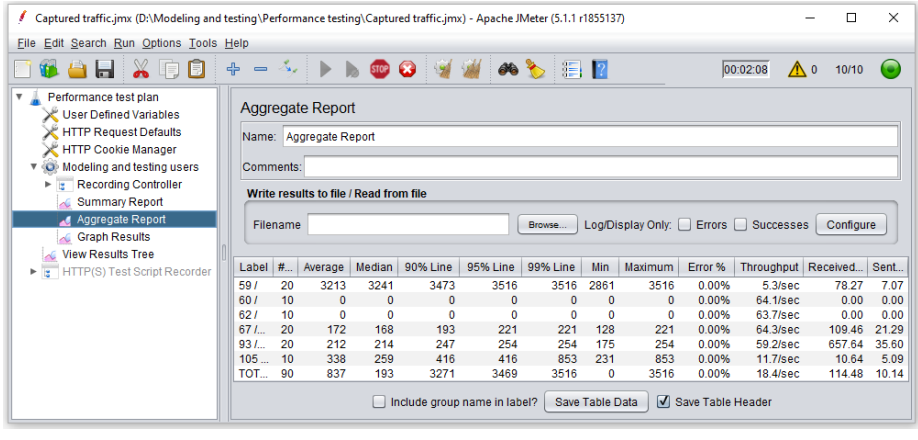


Figure 5. Performance testing with Apache JMeter

II. **Creating HTTP traffic manually with 10 parallel threads:**

1) Create a new Testplan, set the parameters of the *TG* as in the previous exercise in step 3.
2) Add *HTTP defaults* to set the name of the SUT, i.e. the name or IP address of the web server where all HTTP requests will be sent to.
3) Add *HTTP cookie manager* to ensure that each thread gets its own cookies.
4) Add two *HTTP requests*: One will open the main page and one will open a selected subpage of the given website.
5) Add statistics to the TestPlan, validate it, execute test and check the results.

Related tutorials can be found in the JMeter documentation[7]. The examples above are just small illustrations, but if one would like to generate significant

[7]Check chapters *26. Apache JMeter HTTP(S) Test Script Recorder* and *4. Building a Web Test Plan* of JMeter's user manual (`https://jmeter.apache.org/usermanual/`)

amount of load, then it is advised to use the CLI (Command Line Interface) of JMeter instead of its GUI (Graphical User Inerface) for performance test execution and distributing the generation of workload to more computers. Also note that the upload speed of a typical home network may be limiting.

If you are more interested in JMeter, the following list of functionalities may be also worth to check:

- Handling listeners, creating own statistics or html test reports
- Parsing the response with regular expressions
- Testing other types of protocols or other types of load

Wireshark

The free, open source and well-documented Wireshark[8] protocol analyser can be used to augment the functionality of JMeter by observing the protocol stack during performance testing and recording traces for further analyses.

When Wireshark is started, (after selecting the appropriate network connection) it automatically starts capturing network traffic. The recording can be stopped with the red square icon at top left corner and a new recording can be started with the blue shark fin icon.

The tool shows the network traffic in 3 main views (see Figure 6):

I. Packet list pane: shows the main parameters (such as time stamp, source and destination addresses, name of the used protocol, length, further information) of each message. The messages can be sorted by clicking on the corresponding parameter column.

II. Packet details pane: By selecting a message at the packet list pane above, it shows the protocol stack of the given message. Each level of the tree structure of the protocol stack can be folded/unfolded to show the information the user is interested in.

III. Packet bytes pane: Displays the data in hexadecimal view. The given element selected in the packet details pane is highlighted.

Wireshark can filter message exchanges by regular expressions; the corresponding window can be found above the Packet list pane. For example applying `ip.src==192.168.0.0/16 and ip.dst==192.168.0.0/16` Wireshark will display local traffic only, `tcp || udp` filters TCP and UDP (User Datagram Protocol) traffic and `sip.To contains "Bob"` shows packets where the *To* header

[8]Wireshark. Network protocol stack analyzer. `https://www.wireshark.org`

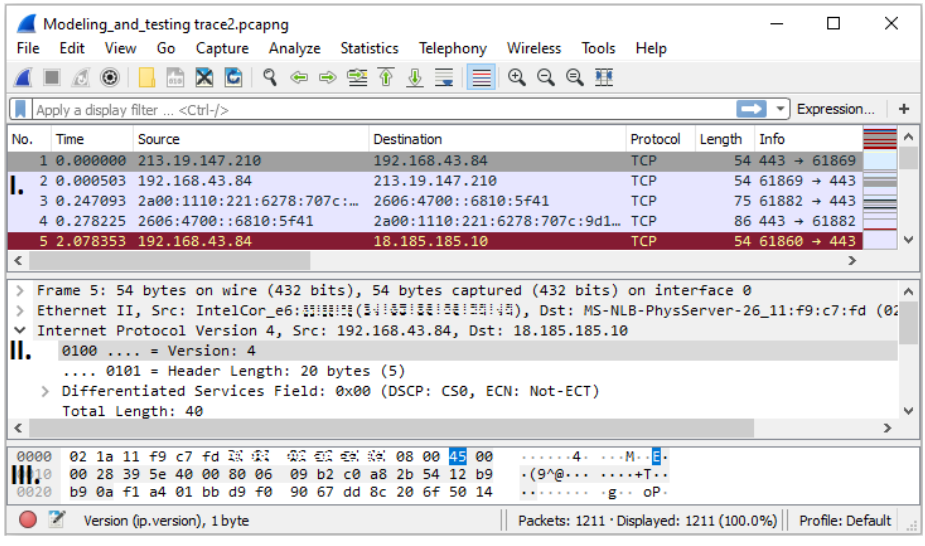


Figure 6. The 3 main views of Wireshark

of SIP messages contain the string "Bob". Wireshark handles more than 3000 protocol types and over hundred thousands of their fields; for more details check the corresponding parts of Wireshark documentation[9].

Note that besides log analyses, Wireshark can be used to load generation as well. To achieve this, a simple load generator application should be implemented that replays the Wireshark trace of a previously recorded traffic[10] with given number of parallel threads (users), schedules the starting of these parallel threads and defines a loop count for each thread. Note that the scheduling of parallel threads can also be done with Jenkins[11], introduced in next section.

[9]Check chapters *6.3. Filtering Packets While Viewing* and *6.4. Building Display Filter Expressions* of Wireshark's User's Guide (`https://www.wireshark.org/docs/wsug_html/`) and `wireshark-filter` of Manual Pages (`https://www.wireshark.org/docs/man-pages/`) to get used to how the filter functionality works. The full list of available filters can be found in the Display Filter Reference (`https://www.wireshark.org/docs/dfref/`)

[10]Note that any selected part of a previously recorded message trace can be saved separately as a new trace, i.e. the test designer can select the part he is interested in.

Jenkins

When the performance tests have been created, one also needs a tool support for the scheduling of different tests and an interface that shows the results of already executed tests. Note that the results of these test executions can also be used as a gating condition to version control systems, i.e. based on the verdict of given test cases, the tool can decide automatically, if a code change is allowed for a commit or not. This automatic gating functionality is essential in the nowadays' fast-paced software development methods (like Agile, Continuous Delivery, DevOps, etc) that are built on top of Continuous Integration. For scheduling, reporting and automating gating of performance tests, Jenkins[11] was selected in our course.

Jenkins is a free, open source, easy-to-use test automation server with good built-in documentation. It has active connections to nodes (servers), it uses script execution on nodes by scheduling, and it is able to show the results of active and archive tests.

Jenkins has many plugins, even some fundamental functionalities (such as changing some basic display properties in test summary pages, integration with Git[12] version control, integration with Gerrit Code Review, etc.) are implemented there. However, not all combinations of these plugins are tested, thus they should be installed one by one and tested in a sandbox first, before they are applied in live environment.

Performance plugin[13] also exist for Jenkins, which can be used to integrate Apache JMeter tests into the pipeline process. The plugin can be used in the following way:

1. Install the plugin (in Jenkins Dashboard select *Manage Jenkins/Manage Plugins*, then click on *Available* tab, search for term "Performance" and select *Install without restart*)
2. Create a new Jenkins job with the *New item* button, then select *Freestyle project*.
3. Open the Jenkins job configuration and add build step *Execute Windows batch command*. Here one can add the path of JMeter binary, Testplan and test report file, respectively with the following commands:
   `C:\<Jmeter_path>\bin\jmeter -n -t`

[11] Jenkins. Test automation server. `https://https://www.jenkins.io`
[12] Git. Distributed version control system. `https://git-scm.com`
[13] Jenkins Performance Plugin. `https://plugins.jenkins.io/performance/`

```
C:\<Jmeter_path>\bin\<name_of_the_test_plan>.jmx -l
C:\<Jmeter_path>\bin\<name_of_the_test_report>.jtl.
```

4. Add the *Publish Performance test result report* under *Post-build actions* to create a diagram from the JMeter test results; here one should define the path of the test report (that should be the same used in the previous step).

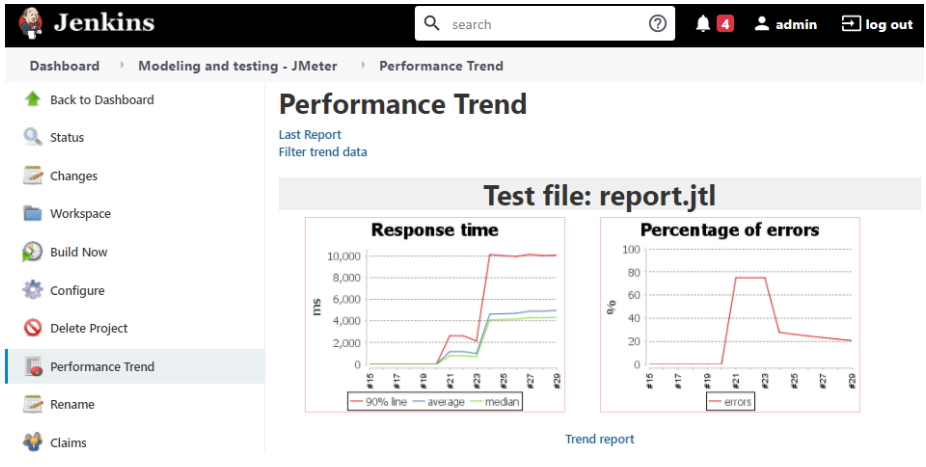


Figure 7. Jenkins with Performance plugin

The result diagram of previous performance test executions are shown in Figure 7. Note that each member of horizontal axis presents the results of a distinct execution, denoted by its build number after the # symbol. More information about using Jenkins Performance Testing plugin can be found in its documentation[14].

Conclusion

During the course, the students learn about the different types of performance testing, various approaches used in load generation and the basics of performance monitoring, reporting and log analyses. The convergence of functional and performance testing is also discussed because in case of large-scale systems it has

[14]JMeter with Jenkins. https://www.jenkins.io/doc/book/using/using-jmeter-with-jenkins/

practically no sense to talk about the functional verification of a product without investigating its performance and vice versa. This is illustrated by a real telecommunication protocol example from the industry that shows how a behaviour model can be created step-by-step for load generation. A tool chain that consists of free and open source tools is also presented to allow students to achieve some basic, real life experiments and skills in the performance testing of websites. At the end of the term, guest lecturers show how performance tests can be applied at a larger scale in the industry both in telecommunication and website domains.

Acknowledgements

The project is supported by the Hungarian Government and co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies).

References

Abbas, R., Sultan, Z., & Bhatti, S. N. (2017, April). Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege. In *2017 international conference on communication technologies (comtech)* (p. 39-44). doi: 10.1109/COMTECH.2017.8065747

Abbors, F., Ahmad, T., Truscan, D., & Porres, I. (2013). Model-based performance testing in the cloud using the Mbpet tool. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (p. 423424). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2479871.2479937

Ahmad, T., Ashraf, A., Truscan, D., & Porres, I. (2019). Exploratory performance testing using reinforcement learning. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (p. 156-163). doi: 10.1109/SEAA.2019.00032

Barros, M. D., Shiau, J., Shang, C., Gidewall, K., Shi, H., & Forsmann, J. (2007). Web services wind tunnel: On performance testing large-scale stateful web services. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)* (p. 612-617). doi: 10.1109/DSN.2007.102

Erős, L., & Csöndes, T. (2010, June). An automatic performance testing method based on a formal model for communicating systems. In *2010 IEEE 18th International Workshop on Quality of Service (IWQoS)*. doi: 10.1109/IWQoS.2010.5542732

ISTQB. (2018). Foundation level specialist syllabus performance testing.

Jiang, Z. M., & Hassan, A. E. (2015, Nov). A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, *41*(11), 1091-1118. doi: 10.1109/TSE.2015.2445340

Koo, J., Saumya, C., Kulkarni, M., & Bagchi, S. (2019). PySE: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)* (p. 136-147). doi: 10.1109/ICST.2019.00023

Németh, G. A. (2016). A finite state machine-based description in performance testing. *UCAAT*.

Németh, G. A. (2020). Teaching model-based testing. *Teaching Mathematics and Computer Science*, *18*(1). doi: 10.5485/TMCS.2020.0469

RFC 3261: SIP: Session Initiation Protocol. (2002). (`https://tools.ietf.org/html/rfc3261` Accessed: 2021-03-16)

RFC 3665: Session Initiation Protocol (SIP) Basic Call Flow Examples. (2003). (`https://tools.ietf.org/html/rfc3665` Accessed: 2021-03-16)

Wu, C.-S. M., Patil, P., & Gunaseelan, S. (2018). Comparison of different machine learning algorithms for multiple regression on black friday sales data. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)* (p. 16-20). doi: 10.1109/ICSESS.2018.8663760

Youness, O. S., El-Kilani, W. S., & El-Wahed, W. F. A. (2008). A behavior and delay equivalent petri net model for performance evaluation of communication protocols. *Computer Communications*, *31*(10), 2210–2230. doi: 10.1016/j.comcom.2008.02.009

GÁBOR ÁRPÁD NÉMETH
EÖTVÖS LORÁND UNIVERSITY, FACULTY OF INFORMATICS, DEPT. OF COMPUTERALGEBRA
H-1117 BUDAPEST, PÁZMÁNY PÉTER SÉTÁNY 1/C., HUNGARY

E-mail: `nga@inf.elte.hu`

PÉTER SÓTÉR
QUALYSOFT PLC, H-1118 BUDAPEST RÉTKÖZ U. 5., BUDAWEST OFFICE BUILDING, HUNGARY

E-mail: `peter.soter@qualysoft.com`

(Received March, 2021)