UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Hendrik Šuvalov

Automating load testing of UXP components

Bachelor's Thesis (9 ECTS)

Supervisors: Yar Muhammad, PhD
Margus Freudenthal, PhD
Ilja Kromonov, MSc

Tartu 2020

Automating load testing of UXP components

Abstract:

This thesis describes an implementation of an automated load tester for Cybernetica's Unified eXchange Platform's components UXP Security Server and UXP Connector. Load testing is putting demand on a system to see how it operates under use. Developing this testing software for UXP's components enables the developers of UXP to constantly get an update on how the system functions and if anything changes in its performance after updating it.

CERCS: P170 Computer science, numerical analysis, systems, control

UXP komponentide koormustestimise automatiseerimine

Lühikokkuvõte:

Töö kirjeldab UXP komponentidele UXP Security Server ja UXP Connector automaatsete koormustestijate implementatsiooni. Koormustestimine on süsteemile koormuse rakendamine, et kindlustada süsteemi funktsionaalsus selle suurema kasutamise korral. Antud töö käigus valminud tarkvara võimaldab UXP arendajatel hoida silma peal süsteemi funktsionaalsusel ning selle muudatustel süsteemi uuendamisel.

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Abbreviations and Terms

DXL – Data Exchange Layer, a technological and organizational environment enabling internet-based data exchange between systems [1]
UXP – Unified eXchange Platform, a data exchange layer technology product developed by Cybernetica AS
UXP Connector – A component of UXP, which enables more liberal use of UXP's services
UXP Security Server – A component of UXP, which follows the protocol of UXP to enable services between other security servers
WSDL – Web Services Description Language, an XML-based interface description language, which is used for defining the functionality provided by a service
GUI - Graphical user interface
CLI - Command-line interface

Contents

1 Introduction

The aim of this thesis is to create automated load testing programs for Security Server and Connector components of Unified eXchange Platform (UXP) on Linux. It will use Apache JMeter, a load testing program, to test how UXP's services function under heavier loads. It will also create human-readable graphs displaying all the relevant results from the tests. When maintaining a data exchange system on a scale as large as UXP's, it is important that all functionalities remain intact even during periods of time with more frequent usage. This can be controlled by load testing the system and seeing how it handles heavier load, making load tests important for production. During load testing, virtual users are created to send multiple requests, in the case of this thesis, in the form of JMeter test files, in order to simulate a real-world application environment.

UXP is a data exchange layer (DXL) which is used across the globe to ease communication between state institutions, organizations and citizens. According to Cybernetica, it is used by approximately 35 million people [2] in countries such as the United States [3], Benin [4], Japan [5] and many others. The usage of UXP for services enables for a fast and secure peer-to-peer data exchange through which it can directly reduce the excessively complicated administrative procedures required to exchange personal data and documents between information systems and the public. Due to the fact that many of the services provided by organizations or state institutions that use UXP are essential for their users (e.g. inventory management by Centers for Disease Control and Prevention, medical care services [3] etc.), it is vital that the system has minimal downtime and none at all if possible which can be ensured by constant load testing of the system.

Section 2 will give an overview of what technologies are used in the implementation of the load testers. It will give a description of UXP and of the functionalities of the Security Server and Connector components.

Sections 3 and 4 will describe the implementation of the load testing software of the UXP Security Server and the UXP Connector.

Section 5 will describe the practical usage of the load testers.

Section 6 will give an overview of the resulting graphs of the software.

2 Technologies used

The implemented load testers were made for UXP Security Server and UXP Connector. They are written primarily in Java and Python respectively. UXP Security Server's load tester is mostly written in Java due to Cybernetica's suggestion, UXP Connector's load tester is written in Python due to the fact that the author is most comfortable and familiar with Python. They are both ran as Jenkin's jobs on a Linux machine. They use Apache JMeter for sending the requests and measuring the performance of the components.

2.1 Unified eXchange Platform

Unified eXchange Platform (UXP) is a data exchange layer (DXL) developed by Cybernetica AS, which enables secure peer-to-peer data exchange between information systems [2]. The system resembles X-Road in its architecture, the first iteration of which was also developed by Cybernetica in 2001 [6]. Its core components are the UXP Security Server, UXP Registry, which maintains the list of UXP members and distributes it to Security Servers, UXP Trust Services, which issues certificates to UXP members and UXP Security Servers and also provides timestamping [6]. In addition to this, there are also four additional components: UXP Connector, which simplifies the implementation of services, UXP Portal, which is a client with authentication and access control for UXP services, UXP Certificate Authority, which handles issuing the certificates to UXP members and Security servers and the UXP Timestamping Service, which certifies the creation time of the requests, ensuring their evidential value [6]. An example of an UXP instance is depicted in Figure 1.

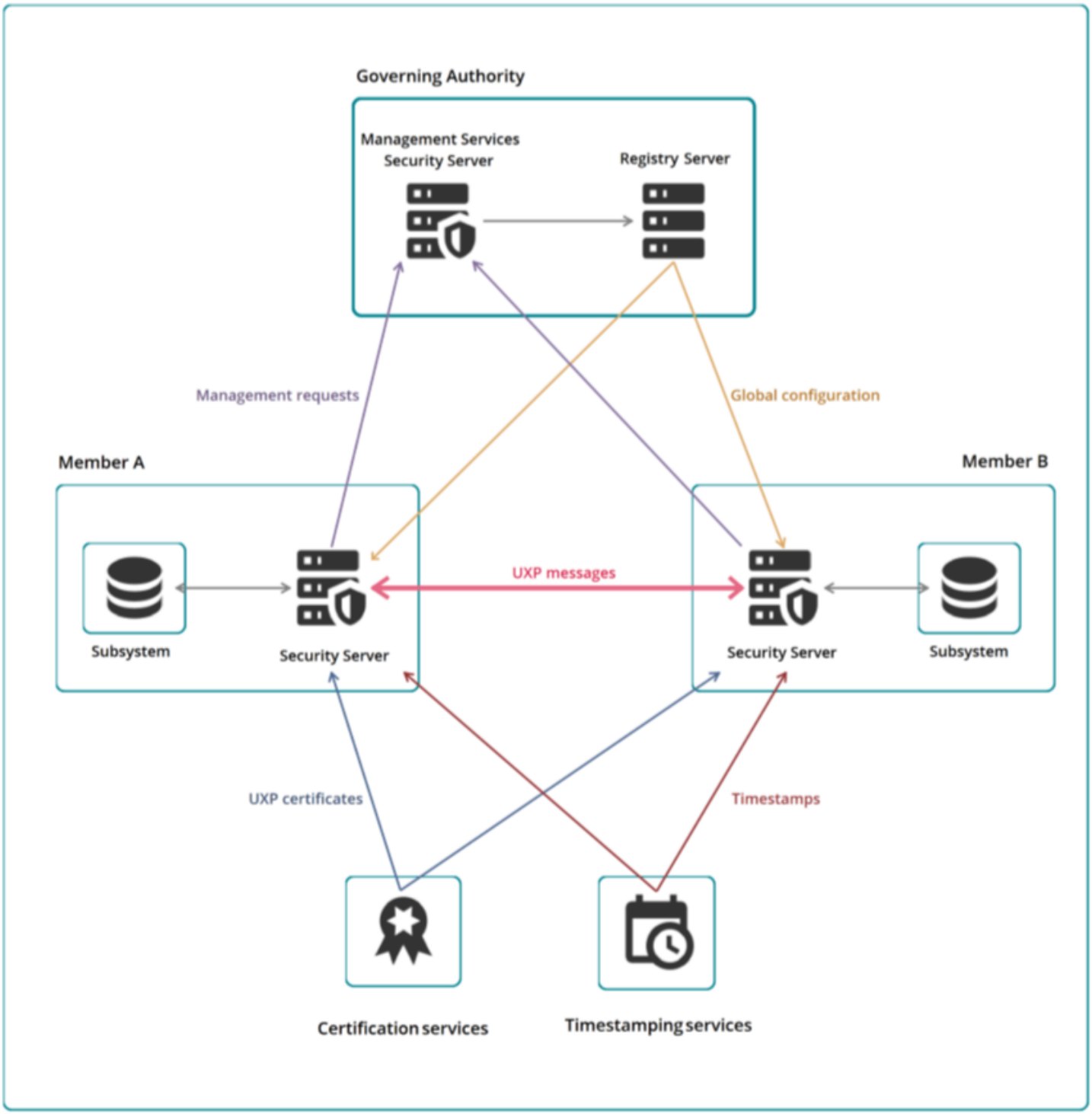

Figure 1. Diagram displaying an example of an UXP instance [7]

UXP instance supports both REST / SOAP services. SOAP messages are structurally based on XML and consist mainly of a header and a body. The headers contain information about the client accessing the service and the service itself. An example of a SOAP request is depicted in Figure 2. Messages sent to the REST API contain a JSON body and have the service and client information in the headers.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xro="http://x-road.eu/xsd/xroad.xsd"
                  xmlns:iden="http://x-road.eu/xsd/identifiers" xmlns:vl="http://x-road.eu/xsd/connector/test11/v1">
   <soapenv:Header>
      <xro:client iden:objectType="?">
         <iden:xRoadInstance>?</iden:xRoadInstance>
         <iden:memberClass>?</iden:memberClass>
         <iden:memberCode>?</iden:memberCode>
         <!--Optional:-->
         <iden:subsystemCode>?</iden:subsystemCode>
      </xro:client>
      <xro:service iden:objectType="SERVICE">
         <iden:xRoadInstance>KYBER_UXP_AA</iden:xRoadInstance>
         <iden:memberClass>COM</iden:memberClass>
         <iden:memberCode>CLIENT1</iden:memberCode>
         <!--Optional:-->
         <iden:subsystemCode>?</iden:subsystemCode>
         <iden:serviceCode>changeServiceCode</iden:serviceCode>
         <!--Optional:-->
         <iden:serviceVersion>changeServiceVersion</iden:serviceVersion>
      </xro:service>
      <xro:userId>?</xro:userId>
      <xro:id>requestid</xro:id>
      <xro:protocolVersion>4.0</xro:protocolVersion>
   </soapenv:Header>
   <soapenv:Body>
      <vl:changeServiceCode>
         <changeInputParameter>changeInput</changeInputParameter>
      </vl:changeServiceCode>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 2. Sample SOAP request

The UXP Security Server mediates requests via services between clients and service providers in a secure manner, using digital signatures and encryption, and in a way that preserves their evidential value [8]. Access to services is handled by the provider on a subsystem or group basis regulated by an authentication certificate so as to remove the possibility of handing out data to unauthorized users [8].

The main function of UXP Connector is to simplify the implementation of services based on an SQL database by allowing the owner of the UXP instance to create services by simply creating an SQL query with variables and connecting it to a database via Java Database Connectivity (JDBC) [6]. It then generates a Web Services Description Language (WSDL) schema of the service and implements it on the instance.

2.2 Apache JMeter

Apache JMeter is a Java application designed to load test functional behavior and measure performance [9]. Load testing is an important process for larger scale systems. It helps the developers see if modifications to the code have affected the overall performance of the system and if the system performs under heavy load (to simulate frequent use in production). JMeter is a tool that enables the user to do this. It supports, among many other protocols, SOAP / REST Webservice testing, which is used in the implementation of the load testers of this thesis. In order to use JMeter, a test *jmx* file is made that contains the request, location of the server, amount of virtual users (threads) to run the test, duration and many other configurable parameters. JMeter then launches the test file and handles creating the threads and sending the requests. At the end of each test, JMeter will provide the results of the test.

JMeter tests can be created in the GUI mode and later activated in the CLI mode, which will run the tests without having to have the GUI open. When creating the tests, the user can put variables in the fields in the format of *$__P(variable name)*. When running JMeter from terminal, the user can then pass values to replace those variables. This allows for extra configuration when running the test, meaning the user can change the duration of the tests, the amount of virtual users and virtually every setting before running a specific test.

2.3 Jenkins

Jenkins is an open source automation server, which is used to automate building, testing and deploying software [9]. A job is created for the software you wish to automatically build and deploy and tasks on how to build and run your software are given via setting the configuration steps for the job.

Source code is usually gotten into the workspace via cloning it from a Git repository. The user can configure exactly which steps need to be taken in order for the build to be ran (e.g. executing shell, invoking a Gradle script e.t.c). If the user does not wish to build the project manually, build triggers can also be set up. The user can choose to build periodically and give Jenkins an interval to specify how often he wants the project to the built. After building and running the code, Jenkins offers post-build actions, which includes publishing it on Git, notifying the user (or other people) via e-mail, archiving the artifacts and many others. For this thesis, the Jenkins job will publish an HTML report as the post-build action, meaning it will save the generated HTML file after every successful build and have it ready on the job's page for the user to view it.

3 UXP Security Server's load tester

The software developed for load testing the Security Server enables the user to configure the tests they wish to run, the parameters of said tests (threads, duration, size of the request) and how frequently the tests are to be run. The software is set up as a Jenkins job, which activates the program automatically and displays the results in the form of graphs in HTML format when it's done.

3.1 Services for UXP Security Server's load tester

The test services developed for the Security Server took into account what might primarily influence the latency of the request. For SOAP services, it is size of the request and whether the content is in the body of the request or outside of it as an attachment. REST service requests were made by taking into account the size of the request's body. In total, three SOAP web services and one REST API were developed along with 8 requests in the form of JMeter test files, which contain the requests and configurable parameters for testing.

The services were made for requests that took into account whether the request is in SOAP/REST, in the case of SOAP messages whether the data is in the body of the request or outside of it as an attachment and the size of the data sent or received, which can be changed by editing *tests.json*. The following requests were made for the developed services:

getBinaryAttachmentSOAP - Binary data in response's attachment

getBinarySOAP - Binary data in response body

sendBinaryAttachmentSOAP - Binary data in request's attachment

sendBinarySOAP - Binary data in request's body

getBinaryREST - Binary data in REST response's body

sendBinaryREST - Binary data in REST request's body

minimalRequestSOAP - SOAP Request with no binary data in response/request

minimalRequestREST - REST request with no binary data in response/request

3.2 Core program for load testing UXP Security Server

The core program is written in Java, which is activated by Jenkins and it first reads in the configuration files from *configuration.properties* and *tests.json*. Then it runs the corresponding JMeter test files, containing the requests, saves the results in the *ResultsCSV* directory as CSV files containing the date of the test and either latency or how many samples were sent in a minute. When all the tests are run, *graphGenerator.py* is activated, which reads in the results of the tests and generates a HTML report using a library called Plotly to generate the graphs. A schema of the work flow is depicted in Figure 3.

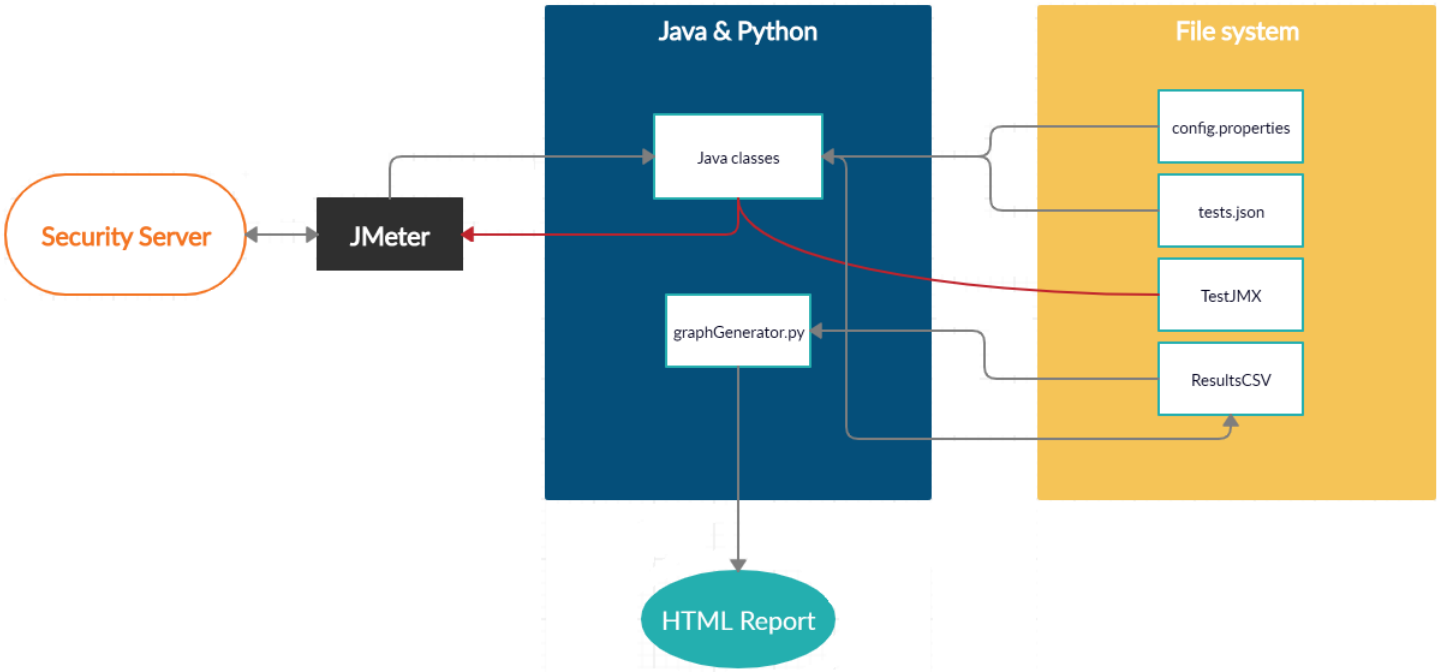

Figure 3. Schema of work flow of components in Security Server's load tester (note that the red color signifies the JMeter test files)

3.3 Configuration files for load tester

Before setting up the software to start automatically load testing, the user has to make the proper changes to *configuration.properties* and *tests.json*. In the *configuration.properties* file, the user has to change the settings for the location of the web services.

configuration.properties has the following properties:

SOAP_WebServerProtocol - HTTP or HTTPS

SOAP_IP - IP or hostname of Security Server

SOAP_PortNumber - Port of the web service path

SOAP_Path - Path of the web service

REST_WebServerProtocol - HTTP or HTTPS

REST_IP - IP of the REST API

REST_PortNumber - Port of the REST API

REST_Path_get - Path to REST service "getBinary"

REST_Path_send - Path to REST service "sendBinary"

REST_Path_minimal - Path to REST service "minimalRequest"

tests.json can be configured to choose the tests the user wishes to run. For every test element, another element can be added with the following parameters:

Threads - How many threads are going to run the test

Duration - How long the test is going to run (in seconds)

For tests that involve either sending or receiving binary data, another parameter is added:

Size - Size of the request/response (in bytes)

4 UXP Connector's load tester

The software developed for load testing the Connector enables the user to configure the tests they wish to run, the parameters of said tests (threads, duration, generated values of requests) and how frequently the tests are to be run. The software for the Connector is also set up as a Jenkins job, which activates the program automatically and displays the results in the form of graphs in HTML format when it's done.

4.1 Services for UXP Connector's load tester

Similarly to the Security Server's load tester, Connector's services took into account what might influence the latency of a request as well, however, different factors apply here from what influences the latency of a Security Server's request. Whereas in the case of Security Server's requests, where only the size of the request and whether the data is an attachment or not matters and the request's or response's structure effect on latency is marginal, in the case of Connector's services, the structure of the request and the response plays into account as well, because they are converted between XML and SOAP format when receiving the request and returning the response. This means that structurally complicated requests (such as nested elements in SOAP request) will take more time due to the fact that it takes more time to convert the request. Because of the conversion, data types also play a role (whether the data is in binary format, integers, e.t.c). The services made for Connector's load tester took those factors into account. Creating the services themselves simply meant creating database functions that generate random data and writing an SQL query for them, such as "*SELECT :dataType FROM db_table LIMIT :size*", where *size* and *dataType* would be substituted with values corresponding to the request. In total, 7 types of requests were taken into account and for each type of request, the data was, in one case, generated and sent by the client and in the other case generated and sent by the Connector.

The requests made for Connector's services varied in what type of data is sent and whether the data is in the body of the request or the response, or outside of it as an attachment. They also took into account the size of the data, which is configurable by changing *config.yml*. The requests made for Connector's services:

longstring - Long string in request/response

flat - Multiple elements in request/response

nested - Multiple elements nested in request/response

BLOBbase64 - Binary large object in base64 format in request/response

BLOBhex - Binary large object in hex format in request/response

BLOBattachment - Binary large object in attachment of request/response

longstringattachment - Long string in attachment of request/response

4.2 Core program for load testing UXP Connector

The core program for load testing the Connector is written in Python. A schema of the workflow is depicted in Figure 4. The requests are generated by Python based on a template request *template.xml* with data for each test generated by what is defined in the *config.yml* file and put into the JMeter *jmx* test files before activating the tests. When the tests are completed, the results are saved in the *ResultsCSV* directory as *CSV* files and the graphs are generated using *Gnuplot* and put into an HTML report. In addition to the latency graphs and the graphs displaying how many requests per minute were sent, Connector's load tester also generates an HTML table, displaying the changes in latency between the last tests and the current ones ran.

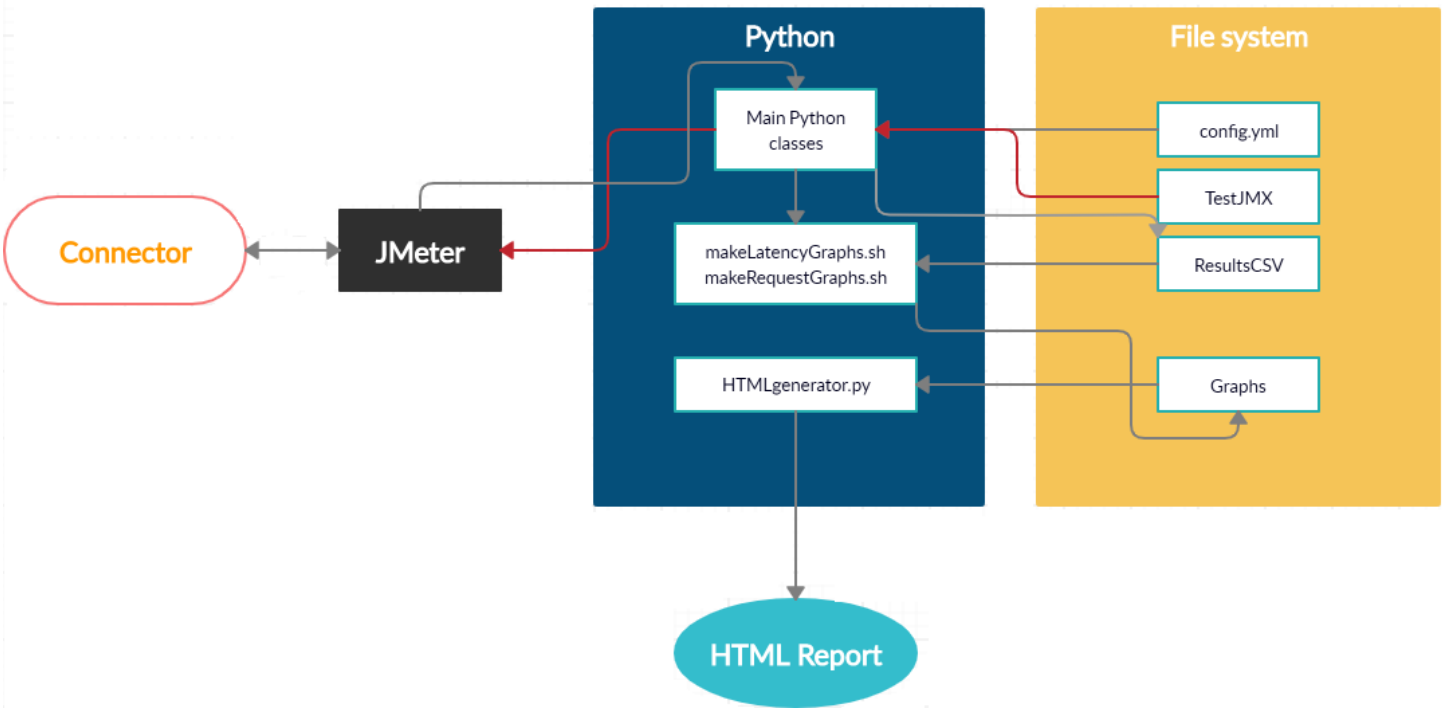

Figure 4. Schema of work flow of components in Connector's load tester (note that the red color signifies the JMeter test files)

4.3 Configuration file for load tester

In order to properly set up Connector's load tester, the user has to change the following variables in *config.yml* in order to set up the usage of JMeter and the location of the web services:

jmeterpath - Path of Apache JMeter on machine

host - IP of the Connector

port - Port of the Connector

protocol - HTTP or HTTPS

path - Path of the services

In addition to these, *config.yml* also contains the parameters for each test to be run:

threads - How many threads are going to run the test

duration - How long the test is going to run (in seconds)

value - Size of the request/response (in bytes) or in the case of *nested* and *flat*, the number of elements to be generated.

5 Practical usage of load testers

Both Connector's load tester and Security Server's load testers are set up as a Jenkins job. They are put into a Git repository, from which the Jenkins job will pull the project into its workspace, run the programs and save the HTML report so that it can be viewed from Jenkins. The reason why Jenkins is useful for this project is that you can configure it to build periodically (e.g. once every 24 hours) and view the resulting HTML report at any given time.

After changing the configurations mentioned in sections 3.3 and 4.3, in order to set up the Jenkins projects, the configurations of the Jenkins's jobs were made. The user has to create a new project, add the Jenkins job corresponding to the load tester to it and make sure the paths to the Git repositories and workspaces are correct. After that, the user can configure at what interval Jenkins will run the builds at. Once that is done, Jenkins will automatically run the programs and generate HTML reports for the user.

6 Results

After succesfully building on Jenkins, results are displayed in the form of HTML reports with graphs as depicted in Figures 5 and 6 below. The report files are generated by Python programs and archived by Jenkins after every successful build.

6.1 UXP Security Server's results

UXP Security Server's load tester generates one HTML report which contains both the latencies of the requests and how many were sent in a minute.

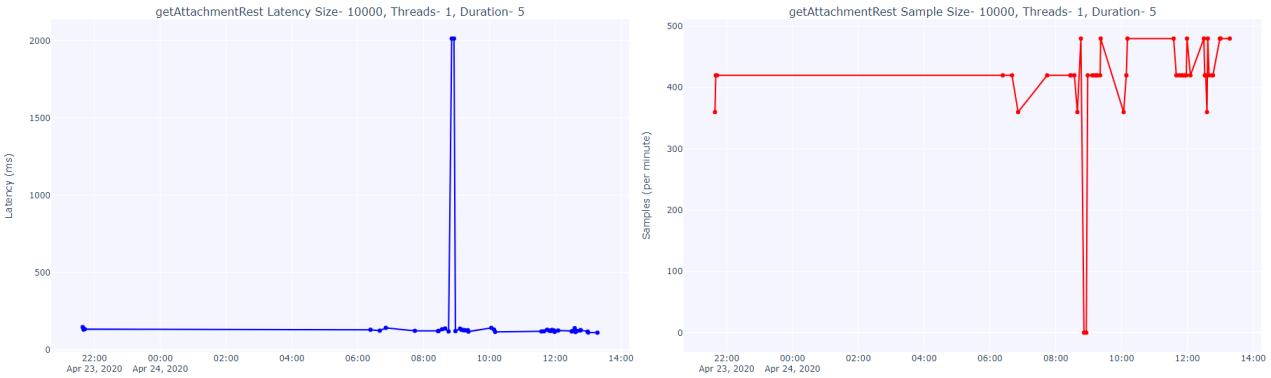

Figure 5. Example of graph in Security Server's load tester

As can be seen from Figure 5, both graphs have the date of the test run on their x-axis, the latency graphs have the tests average latency as the y-axis and the graphs depicting requests sent per minute depict the average amount of samples sent by the tests per minute on their y-axis. Note that the graphs are interactive and individual results will be displayed upon placing one's cursor on a result node.

In the case of Figure 5, if we wanted to analyze the performance of the service based on these graphs, we could bring out that due to the spike in latency and the fact that no requests were successfully sent during one test, the service must not have functioned properly or the server was down.

6.2 UXP Connector's results

UXP Connector's load tester generates 3 HTML reports: one for latencies, one for how many requests were sent in a minute and an HTML table that displays the changes in latencies between the current test and the previous one.

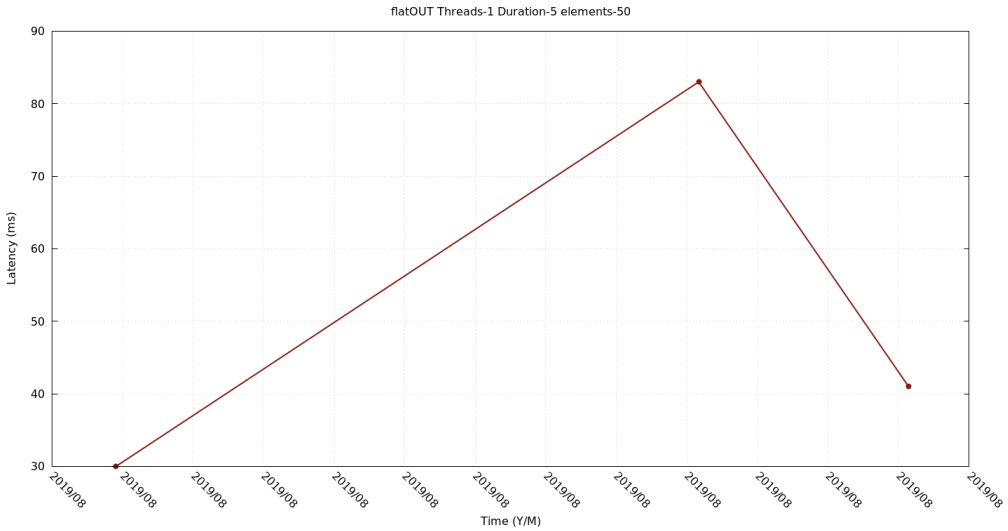

Figure 6. Example of latency graph in Connector's load tester

An example of a graph from the HTML report containing the graphs displaying changes in latency is displayed in Figure 6. The x-axis displays the date at which the test was ran and the y-axis the average latency of the requests. The graphs displaying how many requests were sent in a minute on average are identical, except they have "Requests per minute" on their y-axis.

If we wanted to analyze the performance of the service based on the graph in Figure 6, we could point out that there was a small spike in latency but the service functioned properly over-all.

Conclusion of results

Change in average latency compared to previous build

Test name	Latency	Change in latency
longstring attachment OUT Threads-1 Duration-5 size-10000	75	+0.0%
longstring attachment IN Threads-1 Duration-5 size-10000	39	+0.0%
longstringOUT Threads-1 Duration-5 size-10000	79	-10.2%
BLOB base64 OUT Threads-1 Duration-5 size-100000	122	-12.9%
BLOB hex OUT Threads-1 Duration-5 size-100000	141	-14.0%

Table 1. Example of graph displaying changes between the last test and current one

Connector's load tester also generates a graph displaying the changes in latency for the last test (compared to the one run before that). This makes it easier for the user to see if there has been any drastic change in latency. An example of this is displayed in Table 1.

7 Conclusion

This thesis has given an overview of why it is important to load test the UXP Security Server and UXP Connector components. It describes the implementation of the load testing software developed and what services are used for testing. The thesis also talks about the usage of the load testers and brings out examples of the results that are generated during its usage.

The result of this thesis is in two parts - UXP Security Server's load tester, which is mainly written in Java, and UXP Connector's load tester, which is written in Python. They both use Apache JMeter for load testing and are both automatically built and run using Jenkins. At the end of every build, they produce HTML reports containing graphs of test results which enable the user keep an eye on any changes in performance after implementing changes to UXP and help ensure that UXP functions in periods of more frequent usage.

Future work could be done on combining the two projects into a simpler, unified project and perhaps making the software easier to use over-all. It could also be modified so that it could be implemented on any web service in general and not specifically UXP's components.

References

[1] Data Exchange Layer X-tee. https://www.ria.ee/en/state-information-system/x-tee.html (24.01.2019)
[2] Unified eXchange Platform. https://cyber.ee/products/secure-data-exchange/ (12.12.2019)
[3] Cybernetica News. Centers for Disease Control and Prevention in the United States to Use Secure Data Exchange Platform Built by Center for Medical Interoperability and Cybernetica. https://cyber.ee/news/2020/01-23/ (06.05.2020)
[4] Cybernetica News. Cybernetica Developed the UXP-Based Government Services Portal (Service-Public.bj) for Benin. https://cyber.ee/news/2020/03-26/ (06.05.2020)
[5] Cybernetica News. Cybernetica Develops GDPR-Compliant Data Exchange Between Japan and EU for Medical Data. https://cyber.ee/news/2020/04-20/ (06.05.2020)
[6] Unified eXchange Platform. https://cyber.ee/products/secure-data-exchange/materials/uxp-brochure-2018.pdf (12.12.2019)
[7] UXP Security Server 1.11 User Guide (2019). Cybernetica.
[8] Unified eXchange Platform (UXP) White paper. https://cyber.ee/products/secure-data-exchange/materials/uxp-technical-whitepaper-2018.pdf (07.05.2020)
[9] Apache JMeter. https://jmeter.apache.org/ (02.05.2020)
[10] Jenkins documentation. https://www.jenkins.io/doc/ (01.05.2020)

Appendix

The following Figures 7, 8 and 9 are examples of the configuration files for both UXP Security Server's load tester and UXP's load tester.

```
# SOAP

SOAP_WebServerProtocol = http
SOAP_IP = localhost
SOAP_PortNumber = 8000
SOAP_Path = ws

# REST

REST_WebServerProtocol = http
REST_IP = localhost
REST_PortNumber = 8000
REST_Path_get = restapi/load/getbinary/
REST_Path_send = restapi/load/sendbinary/
REST_Path_minimal = restapi/minimal/
```

Figure 7. Example of *configuration.properties* file in UXP Security Server's load tester

```
{
  "getBinaryREST": {
    "test1" : {
      "threads": 1,
      "duration": 60,
      "size": 10000000
    },
    "test2" : {
      "threads": 1,
      "duration": 60,
      "size": 50000000
    }
  },
  "sendBinaryREST": {
    "test1" : {
      "threads": 1,
      "duration": 60,
      "size": 10000000
    },
    "test2" : {
      "threads": 1,
      "duration": 60,
      "size": 50000000
    }
  },
  "getBinaryAttachmentSOAP": {
    "test1" : {
      "threads": 1,
      "duration": 60,
      "size": 10000000
    },
    "test2" : {
      "threads": 1,
      "duration": 60,
      "size": 50000000
    }
  },
```


Figure 8. Example of *tests.json* file in UXP Security Server's load tester

```yaml
jmeterpath: "/../../../home/hsuvalov/Downloads/apache-jmeter-5.1.1/bin"
host: uxp-hendrik-load.cloud.cyber.ee
port: 4400
protocol: https
path: /service


default:
  threads: [1]
  duration: 5 # Seconds

tests:

  longstring:
    threads: [1]
    duration: 5
    IN:
      include: True
      value: 10000 # Size
    OUT:
      include: False
      value: 10000 # Size

  flat:
    IN:
      include: False
      value: 50 # Number of elements
    OUT:
      include: False
      value: 50 # Number of elements
```

Figure 9. Example of *config.yml* file in Connector's load tester

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Hendrik Šuvalov**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Automating load testing of UXP components,
   (title of thesis)

   supervised by Muhammad Yar, Margus Freudenthal, Ilja Kromonov.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Hendrik Šuvalov
08/05/2020