Generic AAA Architecture Performance testing

Analytical Network Project
Masters programme on System and Network Administration
University of Amsterdam

Authors:

Aziz Ait Messoud aam@chello.nl
Redouan Lahit rlahit@xs4all.nl

Supervisor:

Cees de Laat
delaat@science.uva.nl

Consultants:

Yuri Demchenko demch@science.uva.nl
Bas van Oudenaarde oudenaar@science.uva.nl
Leon Gommans lgommans@science.uva.nl

Abstract

Within the scope of the Analytical Network Project at the Master education System- and Network Engineering[1] of the University of Amsterdam, we conducted a research to the performance of the AAA server technology that represents current implementation of the Generic Authorization, Authentication and Accounting (AAA) architecture being developed by Advanced Internet Research Group (AIRG) at the University of Amsterdam[2]. The report represents general analysis of the major components of the AAA server that is built on the J2EE platform using Tomcat Servlet container. Test model includes variable and controlled time delays on server and client sides that allow indirect components' performance testing. Finally, the report represents voluminous test results and provides summary and recommendations based on the analysis of the test data.

[1] www.os3.nl
[2] www.uva.nl

Table of Contents

1. Introduction

1.1. Background

Currently, after the initial development period and pilot implementation in a few past projects, Generic AAA Toolkits[3] is at the stage when it is ready for the production service in a few on-going projects (e.g., CNL[4], GP-NG, EGEE). At this stage, the performance testing is becoming important issue as component of the quality insurance of the product. Quality insurance provides both guarantee of the technical requirements compliance and feedback for developers which components of the system need enhancement.

A large aspect of application and system architecture is ensuring that a system will perform well, scale to the required number of users, and have minimum down time. These different goals—performance, scalability, and availability—are related, and are based on a set of core concepts that apply to almost any type of system.

1.1.1. Availability

Although "availability" is a critical metric for enterprise systems, it can be defined in multiple ways, which makes it hard to achieve. People frequently classify system availability by measuring the percent of time in which the system is usable, although some measurements do not count planned down time (such as for maintenance). The following table shows these common classes and the associated availability percentages and related annual down time.

Availability class	Availability measurement	Annual down time
Two nines	99%	3.7 days
Three nines	99.9%	8.8 hours
Four nines	99.99%	53 minutes
Five nines	99.999%	5.3 minutes

So for down time measurements it is useful to consider everything from the point of view of the business, and classifying down time as any time the users are unable to access the system. (Note that the down time values given above are based on a requirement of 24/7 availability)

1.1.2. Scalability

Scalability is another key system feature when building applications or infrastructure for the enterprise. Scalability measures how a system's capacity can grow to handle increased load; the upper limit of this growth and the efficiency with which that growth occurs are key defining factors of how "scalable" a system is. In both technical and marketing materials, "scalability" and capacity are often discussed as if they are the same concept; a system that can handle a million users is described as extremely scalable, even if it could never be scaled-down to a reasonable cost to handle 1000 users, or could not be scaled up to two million users.

The efficiency of a system's scalability is determined by the proportional change in hardware requirements as system load is increased. A system that can handle n users with a certain amount of hardware and can increase its capacity to double that number of users by

[3] RFCs 2903, 2904 and 2905
[4] www.collaboratory.nl

doubling the hardware is said to have "linear scalability," which is relatively very efficient growth compared to many enterprise-level systems.

When scaling a system, there are two general ways in to do it: scale up or scale out. Scale-up refers to using a single server, and adding system hardware resources (processors and memory) as needed to increase overall system performance. The other method, scaling out, requires clustering and/or load balancing technologies to support the use of multiple servers, and additional servers are added as required to handle increased load. Certain applications (such as database servers, ERP systems, and others) are best suited to the scale-up model, running on a single server that is made as large as necessary. Other types of systems (such as Web sites and n-tier systems) are able to work well in a distributed environment and can be scaled out across multiple servers.

1.1.3. Performance

Known is that performance and scalability are tightly related and often considered at the same time, it is helpful to separate the two for the purpose of discussion. How well a system performs is usually measured by the amount of time it takes to respond to specific user requests, or to accomplish a specific task. When scalability is included in this definition of performance, we have to modify it to include the number of users/requests being handled at the time of the performance measurement.

1.2. Project description

Over the past years the AIRG[5], have developed an the Generic AAA Architecture.
At the UvA, the group has implemented an AAA server. They have chosen for JAVA Enterprise Beans (J2EE) to build the prototype.
The research group is now faced with questions such as the following:

- What are the maximum load levels that the system will be able to handle in the production environment?
- What would the average response time, throughput and resource utilization be under the expected workload?
- Which components have the largest effect on the overall system performance and are they potential bottlenecks?

During our research we have looked at the performance of the AAA architecture as it is implemented at the UvA.

[5] http://www.science.uva.nl/research/air/

2. The Generic AAA Toolkit server Architecture

The Generic AAA Architecture[6] comprises of four major components, figure 2-1. The three A's in this architecture stand for authorization, authentication and accounting.

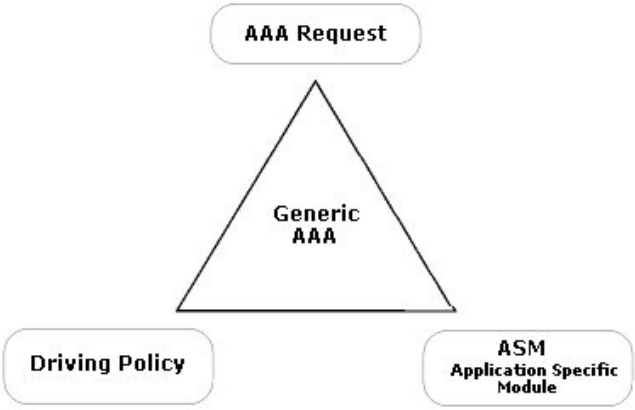

Figure 2-1 : Components of the Generic AAA Architecture

Authorizing a user, to let him use a certain service, can be broken down into three steps: an authorization request, a decision based on a policy and a response or action. A user requesting authorization places his request at an authorization server. The server receives the request and makes a decision on the authorization of the user. This decision is based on the policies the server has for this user or for the service the user requests.

The Generic AAA architecture is based on the three steps the authorization process can be broken down to. The first step in this process is the request. When a user requests authorization, the rule based engine (RBE) of the AAA server receives the request. This engine contacts a database called the policy repository. This database contains all the policies for the services this server offers. A policy may require the verification of external entities, which understanding the semantics of the user request.

The internal world of the AAA server is based upon logic, a variable can be true or false, an authorization reply can be yes or no. Semantics is something the RBE isn't interested in. When a policy requires communication with external equipment to verify an external variable, there's the need for an interface between the logical internal world and the semantic outside world. This link is provided by the application specific modules (ASM's).

When verifying an external variable, the RBE contacts the correct ASM. The ASM then communicates with the external equipment to determine the external variable, using the specific protocol for this equipment. This variable is sent back to the ASM which then replies a logic value to the RBE. The RBE finally decides whether or not the authorization request of the user is granted.

Besides taking care of verifying external variables for granting authorization requests, the ASM's also take care of contacting the requested services. When, for instance, a user is authorized for using the bandwidth of a router, an ASM will contact the router to instruct it to reserve the desired bandwidth for the IP address of the user.

---

[6] Prototype of a Generic AAA Server (Internet-Draft)

Figure 2-2 shows the connections between RBE, the policy repository and the ASMs. The ASM can be build up from a single session bean or a more complex configuration of an entity bean and resource adapter. This resource adapter is used to interface the external resources.

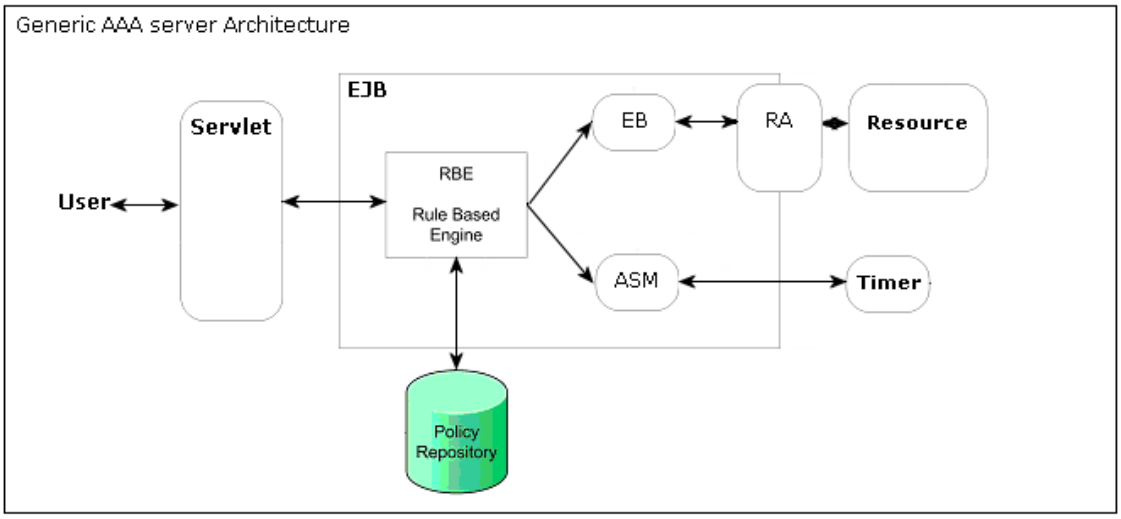

Figure 2-2 : The internal components of an AAA server

2.1. Message Sequences

There are several ways a user can get authorization from an AAA server to use a service provided by a service provider. These ways, which are called message sequences, are the agent, pull and push sequence and are shown in the schematics below.

2.1.1. Agent sequence

In the agent sequence (Figure 2-3), the AAA Server functions as an agent between the User and the Service Equipment (SE). The User sends a Request to the AAA Server (1). The Interface unpacks the Request and sends it to a Rule Based Engine (RBE) (2). Before the RBE will retrieve the corresponding Driving Policy and Reply from the Policy Repository (PR) (4), it asks for a new Session to be created (3). Instructed by the Driving Policy the RBE calls one or more ASM (5) and passes the arguments needed. While an AAA server has exactly one RBE defined, and one Session Manager, it may have multiple ASMs at its disposal. Arguments passed to an ASM may originate from the incoming request or from values returned by previous calls. These arguments might be needed by the Service Equipment the ASM interfaces to (6). Values returned by an ASM (7) may also be inserted into the Reply to the User. Once the Driving Policy has been decided the Reply is returned to the User (8,9). When there is no need for the Session Manager to keep the information of this Session into persistent storage after the User received an answer, the Session Manager might write that information into a log file.

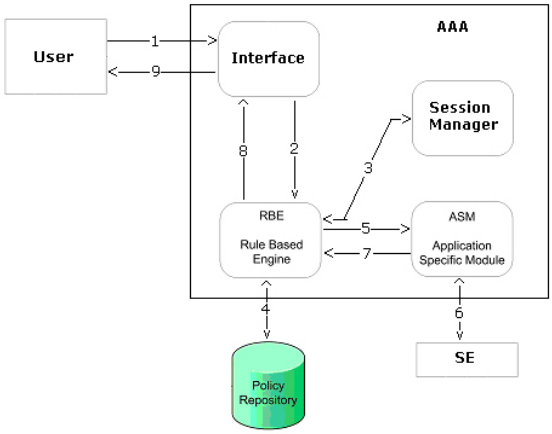

Figure 2-3 : Schematic view of an agent sequence

2.1.2. Pull sequence

With the pull sequence (Figure 2-4), the user directly contacts the service equipment, requesting authorization for a service (1). The service equipment then contacts its AAA server in order to get an authorization for the user (2). The AAA server evaluates the request, applies the policy and sends back an “authorization granted” reply to the service equipment (3). The service equipment sets up the service and tells the user the service is ready for use (4).

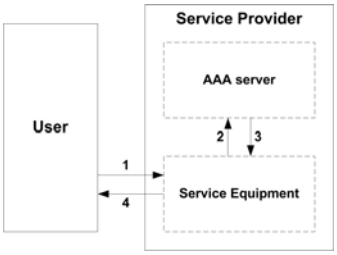

Figure 2-4 : Pull sequence

2.1.3. Push sequence

The push sequence (Figure 2-5) is based on some sort of certification. The user contacts the service provider’s AAA server (1) to get a ticket or certificate verifying that he’s authorized to use the service (2). Subsequently, the user contacts the service equipment and hands over the ticket or certificate he received from the AAA server (3). The service equipment sets up the service and replies to the user that the service is ready for use (4).

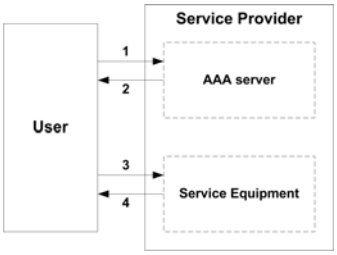

Figure 2-5 : Push sequence

In addition to the sequences shown above, it’s also possible that the organization where the user authenticates is not the same as the organization where the user is requesting a service. This is called roaming. With roaming there is an additional AAA server at the service provider which takes care of the communication between the service equipment and the user.

With a roaming agent for instance, the user requests authorization at an AAA server. This AAA server then contacts the AAA server of the service provider which contacts the service equipment to set up the service. The AAA server of the service provider replies to the first AAA server that the service is ready. Finally the user is notified that authorization is granted and that the service is ready for use.

3. Performance and performance testing

3.1. What is Performance Testing?

Performance testing measures the reliability and responsiveness of the systems under expected and heavy user activity. Performance testing is also used to ensure that the system performs and meets user requirements (and expectations) once released into production.

3.2. Approaches to Performance Analysis

Two broad approaches help carry out performance analysis of distributed systems:

- Load, Stress and Reliability Testing
- Performance Modeling

In the first approach, load-testing tools can be used to generate artificial workloads on the system and measure its performance. Sophisticated load testing tools can emulate hundreds of thousands of 'virtual users' that mimic real users interacting with the system. While tests are run, system components are monitored and performance metrics (e.g. response time, latency, utilization and throughput) are measured. Results obtained in this way can be used to identify and isolate system bottlenecks, fine-tune application components and predict the end-to-end system scalability. This approach can be expensive, since it requires setting up a production-like testing environment to conduct the tests. Moreover, it can not be used in the early stages of system development when the system is not available for testing.
The three different tests can be used to:

- Load Test: simulates real world traffic and activity for the application under lest. Throughput, stability, and responsiveness of the application are measured against expected or required metrics.
- Stress Test: used to determine the breaking point of the application under intense conditions. For example transactions are sent to the server as quickly as possible to saturate the application. This test is useful to determine not only when the system will break, but also the maximum number of requests per time metric (requests per minute) the system can handle.
- Reliability Test: determines how long the application can sustain optimum performance levels under expected loads. This particular test places a steady or consistent workload on the application for a considerable period of time.

In the second approach, performance models are built and then used to analyze the performance and scalability characteristics of the system under study. Models represent the way system resources are used by the workload and capture the main factors determining the behavior of the system under load. Performance models can be grouped into two common categories:

- simulation models
- analytical models

Simulation models are software programs that mimic the behavior of a system as requests arrive and get processed by various resources. The structure of a simulation program is based on the states of the simulated system and events that change the system state. Simulation programs measure performance by counting events and recording the duration of time spent in different states. The main advantage of simulation models is their great generality and the fact that they can provide very accurate results. However, this accuracy comes at the cost of the time taken to develop and run the models. Usually lots of long runs are required in order to obtain estimates of needed performance measures with reasonable confidence levels.

Analytical models are a cost-effective alternative to simulation. They are based on mathematical laws and computational algorithms used to generate performance metrics from model parameters.

Each test will allow the tester to deliver a complete and thorough analysis on the performance of the application under test, and identify bottlenecks that prohibit performance gains.

3.3. Typical application performance problems

During the research we determined that there are a wide variety of problems that can surface during the application lifecycle. For J2EE Web applications in production, user experience of performance is affected by many external network infrastructure factors that are independent of application behavior. External monitors can improve isolation of problems, assisting triage. From the J2EE application diagnostic perspective, it is essential to be able to capture and correlate specific external parameters, such as HTTP arguments, that can drive performance problems in a J2EE method or sequence of transactions. Specific latencies and parameter captures are also frequently needed to identify problems at J2EE interfaces to external systems, such as backend databases, legacy systems, and packaged software.

Within the J2EE environment, some of the most common problems include:

Code problems:

a. Slow methods
    - Consistently slow methods
    - Intermittently slow methods, related to specific user/data values driving problematic application behavior
b. Synchronization problems, including both under synchronization and over synchronization for locks and threads
c. Memory problems, including memory thrashing and memory leaks
d. Coding practices, such as using exceptions as a means to transfer control in the application

Application Server configuration problems:

a. JDBC Connection pool size
b. JVM Heap size
c. Thread pool sizes
d. Application component cache size
e. Message queue buffer size
f. Message queue persistence

Architecture and design problems, with a wide range of issues, such as:

a. Data marshalling problems resulting from filtering at the wrong tier
b. Single-threading resulting from inadequate synchronization design in custom code

Workload:

a. Number of clients
b. Client request frequency
c. Client request arrival rate
d. Duration of the test

Physical resources:

a. Number and speed of CPU(s)
b. Speed of disks
c. Network bandwidth

Application specific:

a. Interactions with the middleware
    - use of transaction management
    - use of the security service
    - component replication
    - component migration
b. Interactions among components
    - remote method calls
    - asynchronous message deliveries
c. Interactions with persistent data
    - database accesses

An effective diagnostic toolset must provide capabilities and techniques to be able to isolate and identify the root cause of each of these common problems, regardless of when they emerge, from development through production.
Capturing J2EE performance data sufficient for solving this range of problems is a significant technical challenge.

4. The test design

In this section, we introduce our approach to performance testing of the AAA Architecture. Our approach compromises a performance testing process that consists of the following phases:

1. First we had to select/define a basic use-case scenario relevant to performance, given current AAA architecture design.
2. Create a program that provides necessary functionality to test major components affecting the performance
3. Define the variables and the range of their variation to simulate typical usage of the AAA Server. - Adjustment to already existing components so the use-case could be carried out.
4. Execution of the test, including: generate workload, initialization of the persistent data and reporting of performance measurements.

4.1. Basic test case

Chapter 3.3 classifies the main parameters relevant to performance testing of distributed applications. First, important concerns are traditionally associated with workloads and physical resources, e.g., the number of users, the frequencies of inputs, the duration of the test, the characteristics of the disks, the network bandwidth and the number and speed of CPU(s).

When designing our test use-case we decided to not take the performance of some components in consideration, we just concentrated on the next values:

- The http connection pool size of Apache and tomcat
- The Servlet
- The size of the EJB container
- The ASM timer

During our test we didn't evaluate the influence of the network, the CPU speed or the Database. Or one of the other components that are listed in chapter 3.3.

Based on the provided Generic AAA Architecture (Figure 2-2), we decided to use the next use-case (Figure 6-1):

- User sends a message (SNBmessage.xml), with some attributes. In our case we just used the timer condition attribute.(step1)
- Apache/Tomcat receives the message and then it forward it to the Servlet (step2)
- The Servlet control's if the attributes are valid, then it sends the request to the EJB (step3)
- RBE receives the messages (in the current implementation the RBE also checks if the attributes are valid or not), then is controls the policy in the policy repository (step4), based on that policy it starts a Session Bean which acts as an ASM (step5).
- The ASM checks the attribute(timer condition) and then starts a timer (which simulates an application) this timer uses the timer condition attribute (step6).
- When timer reaches 0 (step7), then the ASM gives a response back to the RBE (step8)
- The EJB gives answer to the Servlet (step 9)
- Servlet gives answer to apache/tomcat (step10)
- User receives answer (step11)

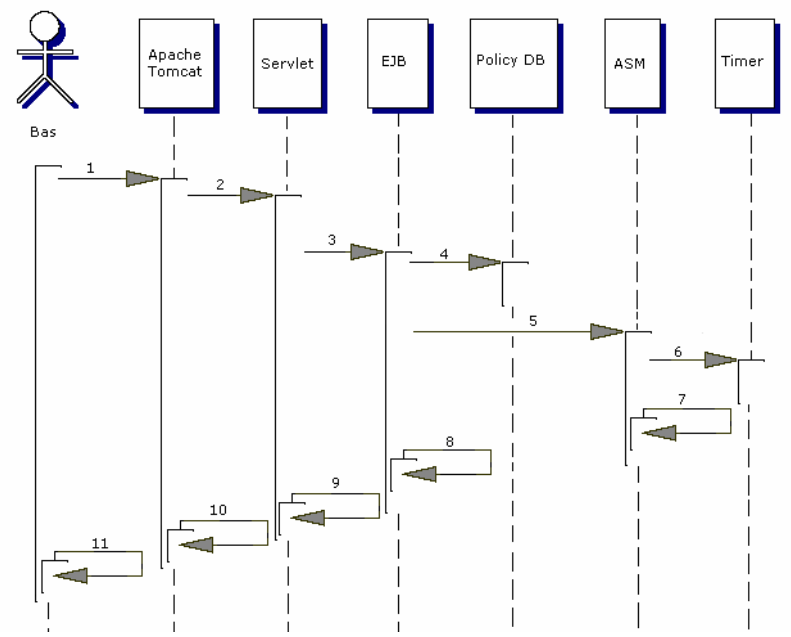

Figure 6-1 : test use-case

To simulate the use-case we had to develop a JAVA program, which was capable of sending different amounts of messages with different arguments.
It was also necessary that this program record the response and the data received from the AAA server. The final version of this program is available in Appendix B.

4.2. The java test program

From different machines (locations) we send SOAP requests to the AAA server. We monitor and analyse how the AAA server reacts on different amount and different types of requests. When the AAA server receives a request it has to process it. We designed a java test program to send a large number of requests at the same time. To get better results we had to make some adjustments in de Servlet code and to make better analyses we had to modify some parameters of tomcat on the server a couple of times. In the coming sections we will discuss there tests and the results.

Roughly speaking our Java test program works as follow:

- Test program starts with two given parameters. (First is a XML file and second is a Integer).
- SOAP message is generated.
- Thread start internal his timer before sending out the SOAP message given as input argument (Number of threads is the second argument of the test-program input).
- SOAP response comes back an the timer stops.
- Value of the timer and the value of the SOAPAction header (see next section) are saved in separate files.
- Test program stops.

4.3. The test setup

To make accurate analyses we needed much data from different machines. These machines where connected through three different networks.

Machine name	CPU	Memory	Other information
Azizlap	AMD Athlon XP 1,5Ghz	700MB	using cable to connect to the internet (Chello) with 1500 Kbit/sec connections speed.
Redlap	Intel Celeron 2.4 Ghz	512MB	using ADSL to connect to the internet with 2240 Kbit/sec connections speed.
Mac-G5b	Dual PowerPC 2Ghz	4GB	100 Mbps, UvA network
Mac-G5c	Dual PowerPC 2Ghz	4GB	100 Mbps, UvA network

To ensure that the results we have found are not randomly and it was not a coincidence, we conducted the same test at least two times. From 2 different places.

4.4. Variables Adjustments

4.4.1. Servlet and SOAPAction header

Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. We added small extensions to the Servlet on the AAA server. We let the Servlet start a timer when a SOAP request, that has been received from the requestor, is forwarded to the EJB container. When a response comes back from the EJB container the Servlet stops the timer. The measured time (in milliseconds) is sent back to the requestor together with the actual SOAP response. We send this measured-time value back in the HTTP header and not in the SOAP response body. Therefore we use the SOAPAction header. This header is meant for other proposals but we use it to achieve better test results without creating more overhead.

We could have chosen to send the measured-time value back to the requestor in the SOAP response body, but that would mean that the AAA Servlet must modify each SOAP response before it is sent back to the user. This would increase the response time of the AAA server because in this case each response must also be processed backward at the user side. By using the SOAPAction header we achieve the same goal with minimum adjustments and overhead.

4.4.2. Thread and sleep time

We use threads in our Java test program to send multiple SOAP requests to the AAA server. This way we are able to simulate a DoS attack. The “sleep” time is the time (gap) between SOAP requests. This value must be chosen carefully, it may for example not be larger than the live time of a java bean that is been initialized by a request on the AAA server. If the sleep time is larger than the live time of a java bean, the thread effect will be lost and the AAA server will be able to process a request before another one is arrived. One of our goals was to monitor how the AAA server reacts when it receives a large number of requests.

The sleep time value may also not be to small. Each thread need some time to start correctly. When the sleep time value is to small threads could conflict together when they access the resources (log files for example) of the machine they run on. We have chosen 2000 ms for this value but our java test program was also stable with 1000ms. How ever the AAA server did not react differently on our test requests, this could be useful for future tests.

4.4.3. Delay

Each process at the AAA server takes an amount of processing time. After the SOAP response is sent back to the user, the process dies and the recourses, that where in use by the process, became available to other processes. The longer a process lives the longer the resources are occupied. When the maximum of processes is achieved the AAA server may react by ignore other incoming processes (or by dropping them) or by let them wait a while. This is because we are dealing with a pool-mechanism, which handles the user request. Which takes a resource from this pool and put it back into the pool again when it is ready.

The way the AAA server react depends on which resources are occupied. The processing time depends on the type of request. In our test we send a xml file in our SOAP request with a value Timer Condition. Whit this value we can define the processing time of the request on the AAA server. The Time Condition value is 1 or greater.

Note: The values of Time Condition are not seconds but time units.

4.4.4. The EJB container and its limitations

Central to the J2EE specification is the Enterprise JavaBeans (EJB) framework. EJBs are server-side components, written in Java, that typically execute the application business logic in an N-tier application. An EJB container is required to execute EJB components.
The container provides EJBs with a set of ready to use services including security, transactions and object persistence. Importantly, EJBs call on these services declaratively by specifying the level of service they require in an associated XML file known as a deployment descriptor. This means that EJBs do not need to contain explicit code to handle infrastructure issues such as transactions and security.

An EJB container also provides internal mechanisms for managing the concurrent execution of multiple EJBs in an efficient manner. EJBs themselves are not allowed to explicitly manage concurrency, and hence must rely on the container for efficient threading and resource usage, including memory and thread usage for application components (EJBs) and database connections.

EJB containers typically allow the explicit configuration of the number of threads that the container uses to execute application EJB code. This configuration therefore represents an important tuning option. Too few threads will limit performance by serializing much of the application processing. Too many threads will consume resources and increase contention, again reducing application performance.

The EJB-container has its limitations. This limitations reside in values that must been chosen carefully. The maximum Pool Size value sets the maximum EJBs that fits in the container. This is the maximum of EJBs that can be initialized. If this maximum is achieved and other request comes in than this new requests must wait. They cannot wait forever because after a certain time the exceed the request is been dropped and a timed out is generated.
This could be caused by the exceeding of the TTL time in HTTP but it is also possible that it is the SOAP layer which implements the client-server connection is being timed out.

Other EJB-container values are Steady Pool Size, Pool Resize Quantity and Pool Idle Timeout. For the exact meaning of this values and other we refer to source paragraph at the end of this report.

During our test, we tried to find out what the effect of changing one of the default settings of the EJB container.

4.4.5. HTTP connection pool

As we mentioned in the previous section, changing a value may effect other processes or cause unpredictable behaviour. For example, with the right user right and knowledge the time to live (TTL) of a HTTP connection can be decreased. When this value is not sufficient there will fail more EJB requests when the container is full and the requests must wait (see previous section). Therefore changing parameter must also be done in a correct sequence.

The HTTP connection pool sets the number of connections may be accepted/running by tomcat (the web server) at the same time. This parameter is called acceptor-thread. Unfortunately because we run out of time we could not test the effect of the other parameters of tomcat. We think that it would be very useful to do tests with this parameters in the future.

5. Test results and analyses

In this chapter we will discus the results of the test and we will try to give analysis to what we have found.

5.1. Charts and data representation

With the data which we gathered from the tests we can generate different charts. When we look at this charts we see patterns. The time between sending a SOAP request (from a user machine) and receiving a SOAP response we call Total Time. EJB Time represents the time the EJB needs to processes a request.

We can generate the charts on two ways. With the use of Microsoft Excel or with a PHP script. The PHP script uses a MYSQL database which is filled from the Java test program. Creating a chart in Microsoft Excel is done manually. We see interesting things in the most charts. We see patterns, glitches and inexplicable (not with our tests) behaviour. We now will discus some of the charts.

In most tests we see that the total request-response time is stable.

5.2. Example 1

In this test we sent 2000 SOAP requests from 4 different machines. For all requests we choose delay timer (discussed in 4.4.3) 1 and all other settings (see chapter 4) where the same for each machine. The charts of the 4 machines looked as showed in figure below.

When analysing the charts we have discovered that there is a certain patterns that are interesting, as you can see in Figure 7.

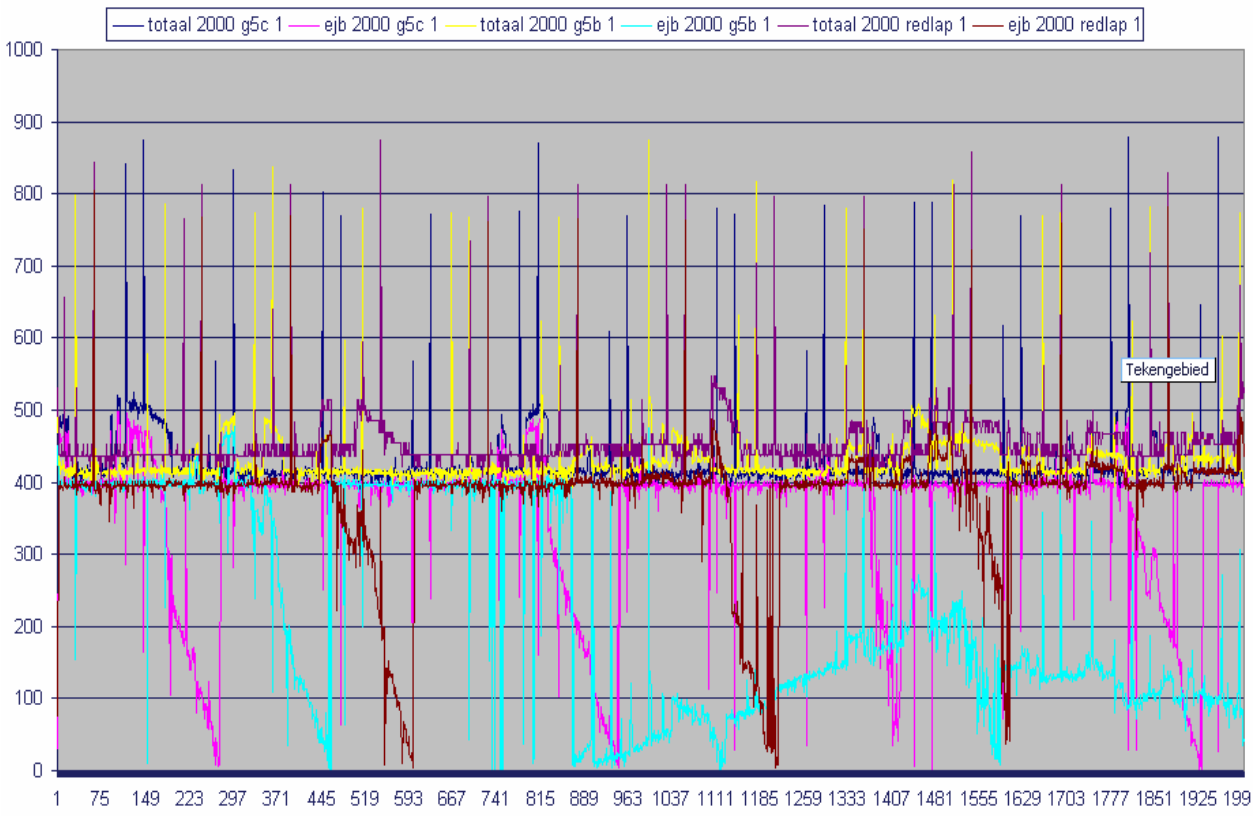

Figure 7: Response times from different machines

To be sure it was not a coincidence, we decided to do this test several time. As you can see in Appendix A (Figure 12, Figure 13,), we can be certain that this is not randomly and was not coincidence.

To discover the trends, we used the floating average function $F_t = \frac{A_t + A_{t-1} + \ldots A_{t-n+1}}{n}$. After using the floating average function in MS Excel we could create a more clear and glitches free charts Figure 8. Where we can see the pattern much better.

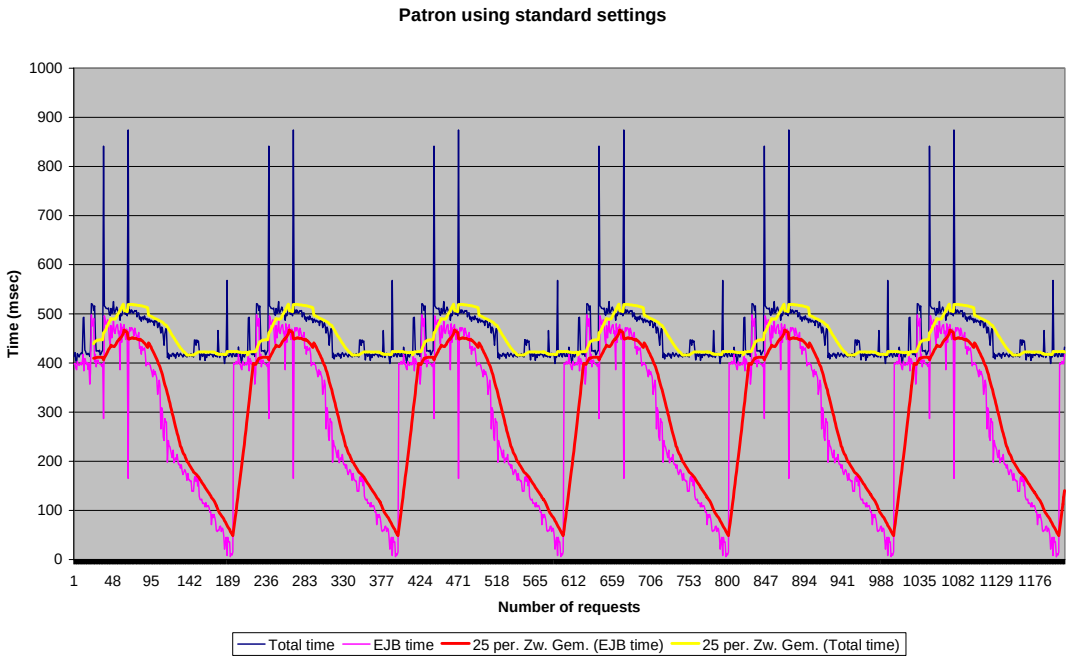

Figure 8 : Pattern test 1

During this test the maximum Pool Size of the EJB parameter was set to 32 and HTTP acceptor-thread parameter was set to 100. Both are default values

As you can see in figure 8, three things are quite remarkably. First even when the EJB processes a request in a shorter time the Total Time remains stable (Yellow line), this applies for all the request from each test machine. The only reason we can find for this, it maybe has to do with the Servlet settings. This could indicate that the Servlet holds the response a certain amount of time, and then it send it back to the client.

The second we recognize a pattern in the EJB Times (the Red line). The EJB Time of each test machine raises and then drops for a while and then it raises suddenly to plus minus 400ms. This pattern repeats itself during plus minus150 requests. This could be caused by a combination of the HTTP acceptor-thread (see 4.4.5) and the way the EJB processes the requests. The Apache/Tomcat accepts 100 requests, it forward them to the EJB. Then the EJB starts Session beans. After the sessions are closed, then Apache/Tomcat accepts other 100 requests and sends them to the EJB. This could indicate that the first raise of the time has to do with the start time of the sessions. And after a while the EJB starts to add more Session beans, which leads to dropping. So it could reuse the already initiated object to accept the new calls for processing. First raising means building up the queue, initiating the first round objects. And because the timer condition is set to 1. When the sessions are closed the EJB receives 100 requests from Apache/Tomcat and then the pattern starts again.

5.3. Variables Adjustments

To explain the above theory we decided to conduct two extra tests. The first test we change the HTTP acceptor-thread from 100 to 1000, and keep all the other setting stable. The second test we change the EJB container to 64 and kept all other setting the same.

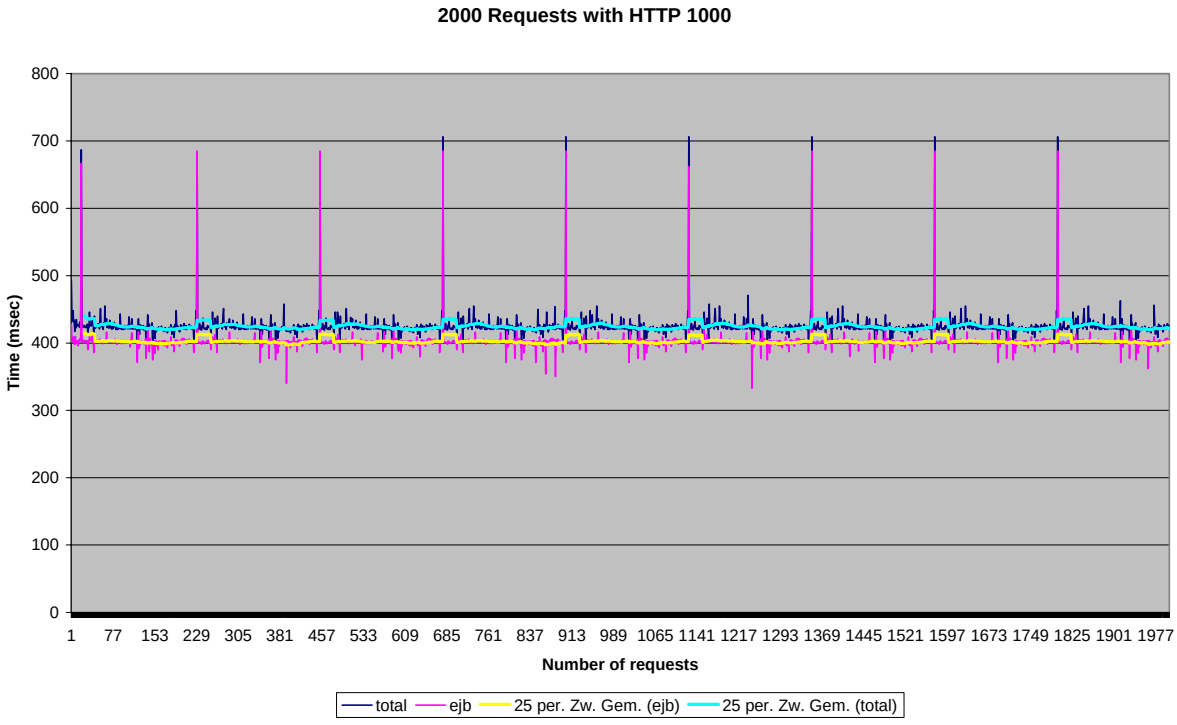

Figure 9 : 2000 Requests with HTTP sets to 1000

5.3.1. HTTP acceptor-thread

As we can see in see Figure 9, when we change only the HTTP acceptor-thread we notice that the EJB time is almost stable, the only thing we can see is that there is another pattern, which repeat its self in plus minus 200 requests. There we can see that the EJB processing time is getting higher and then it drops and gets stable. The most interesting in this part is that we can see that EJB time is the highest during plus minus 30 requests. This almost the same amount of the session that are started simultaneously in the EJB container (32). Therefore we can assume that this peak is the start time is needed to start the EJB sessions.

5.3.2. EJB container

In the second test we can see that the adjustment of the EJB container has no big impact on the processing time (Appendix A Figure 14). But instead, we can clearly see that the HTTP acceptor-thread has a bigger impact on the EJB processing time, then the size of the EJB itself.

After using the floating average function (Blue and Yellow line) we can see a clear pattern (Figure 10). We detected a certain pattern which repeats its self in plus minus 100 requests. Again this results allows us to suggests that this has to do with the HTTP thread-acceptor settings which is set to 100.

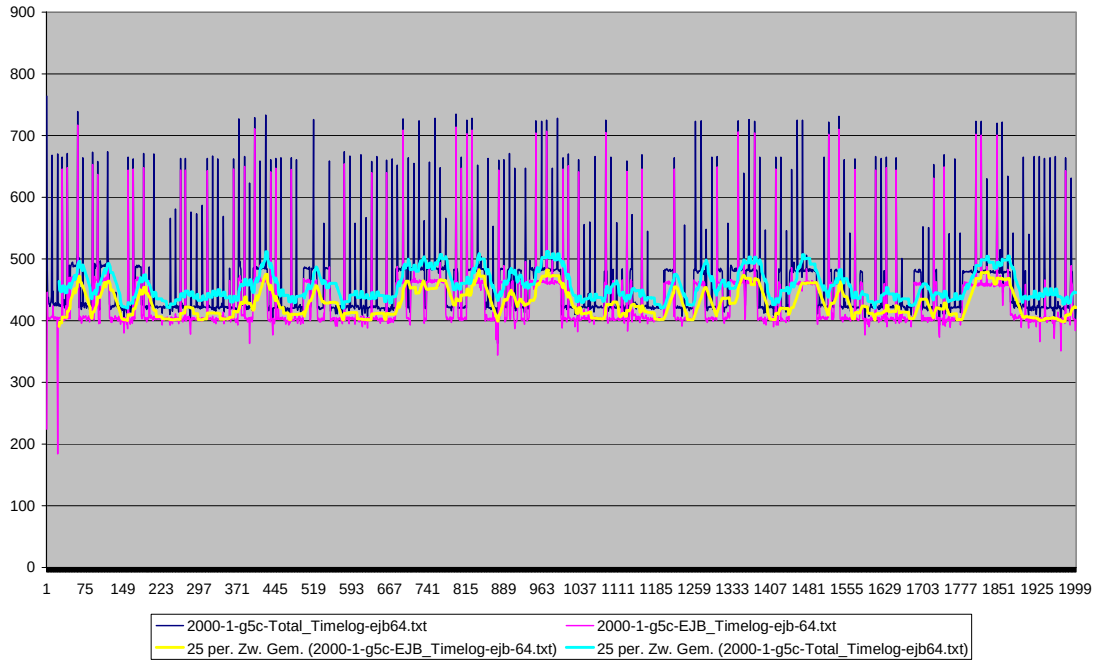

Figure 10 : Floating average with 2000 requests accepting 100 HTTP connections

Based on the above results we can assume that when using pooling mechanism the EJB has a linear performance.
At the end of this series of tests, we decided to see what the effect will be if we change both the values(HTTP thread-acceptor and EJB container size). As we can see in (Appendix A Figure 15) the response time is constant and does not show any major changes.

5.3.3. Changing the timer

During our experimental testing, we discovered that the value of the timer has the greatest impact on the processing time of the EJB. But also on the performance. It was for us obvious that when we change the timer to a higher value that the processing time well increase with the same value of the timer. But while testing we discovered that when sending a timer

higher then 5, the number of responses drops dramatically. It had even the effect that the Servlet was crashed and we did not receive any good response.
After a couple of experiments we found that we can send up to 50 requests with a maximum timer of 20 and still receive good responses, but when we send more requests is fails. Most of them get lost or return a SOAP error. This is shown in Figure 16 and Figure 16 in Appendix A. The errors obviously came from the EJB and not from the tomcat or the AAA Servlet. Therefore we assume it has something to do with the delay timer we changed. Some machines received HTTP connection timed out errors in between.

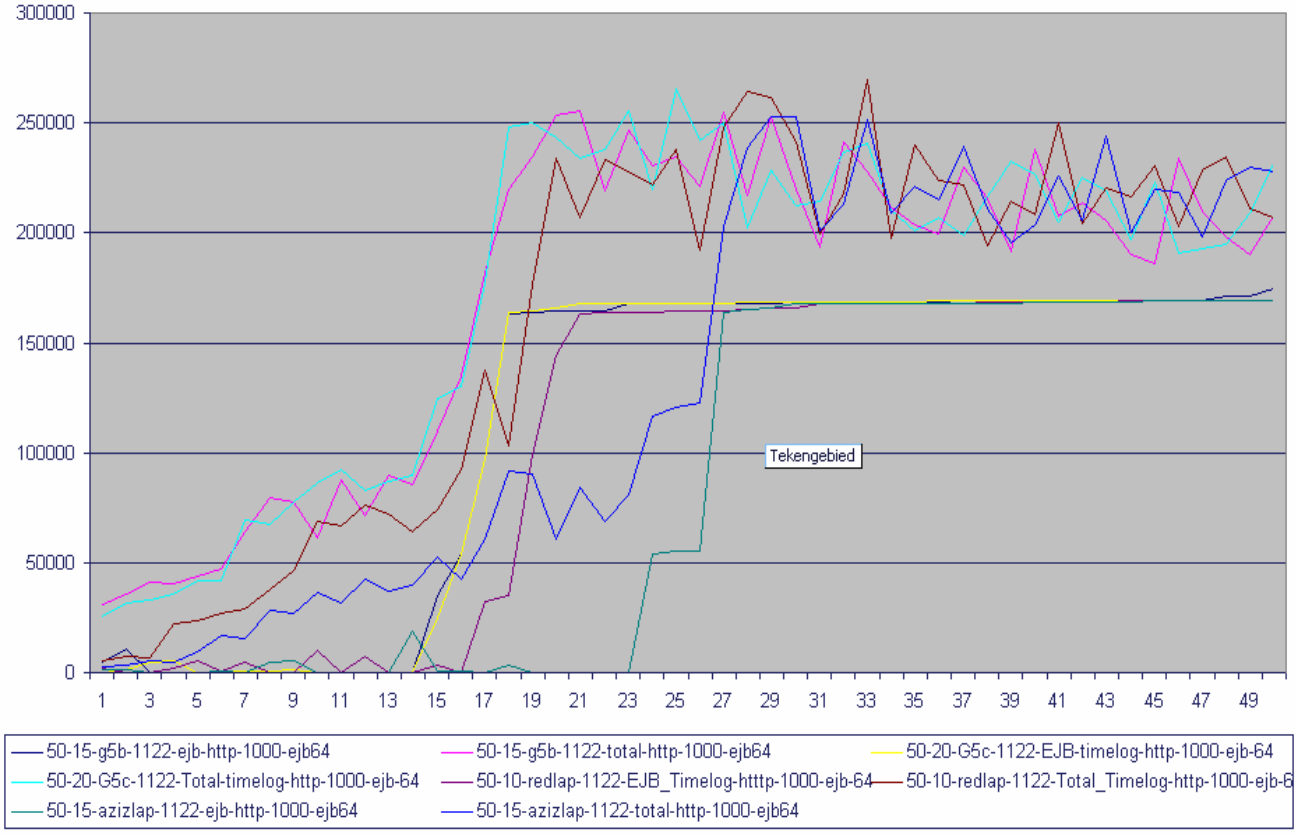

Figure 11 : 50 requests with higher timers

As we can see in Figure 11, we noticed that each chart first climbs slowly and than it reaches a stable point and it stays stable. Secondly we noticed that in the part where all of the charts climbs, Total Time charts do climb faster than the EJB Time charts. EJB Time charts first stay low and stable but they climb faster.

Unfortunately we could not explain why the server behaves like this. We tried to see if this effect has anything to do with the processing time inside the EJB, but we have failed.

Therefore we decided to send a request, while the EJB still processing the first one. What we did was try to find how long each process take. We know that the requests are send every 2 seconds. We know that when using timer 1, that the average processing time by the EJB is 400 ms. And the average total processing time was 500 ms. So together with Bas we decided to set the EJB container size at 2, and also not to allow any other connections. And set the maximum thread-acceptor in HTTP to 10, so it can not accept and forward more then 10 messages. In these case means that only 2 requests will be successful and the others will be dropped.

We tried to calculate how long 1 time unit is actually, based on the responses we got from early tests, we assumed it was about 2 seconds.

So we send 10 request using timer 50 (which we think it has the same value of 100 sec) and assuming that the processing time of the EJB will be at least 100 sec and. We were expecting that the EJB will only process the first 2 request and the other 8 will be dropped, amazingly and totally unexpected the EJB expected and processed all the requests.
And after many efforts to find out what the reason that causes this, we failed and finally we gave up.

Afterwards we realize that this could have something to do with the timer initialization which is random. And therefore 1 time unit has never the same value.

5.4. Summary

We used our use-case model to predict system performance in several different configurations varying the number of the HTTP connection that are allowed, the number of the requests, the size of the EJB container and the timer conditions.

After all the tests we made, we can say that the results can not give us clear answers on the questions we started with in section 1.2. But we still can make some summary's.

With the results from our test we are now able to dynamically monitor the behaviour of the AAA server. We have made different tests and we described future tests. To get statistics there are two ways. We can process data with Microsoft Excel and we have made a PHP page that is connected to a MYSQL dataset where the same results are as in the files. Although we were very motivated we had to limit our test because of the lack of time. But despite of this we have achieved our goals and received some interesting results.

When we attacked the server with a large amount of different request from different machines we saw that the server reacted very strange. We did this a couple of time and the results were different. In some cases the database connection was lost and in other cases we got only HTTP connections timeout errors of all the machines. After a lot of testing we saw that the HTTP connection pool of Tomcat (discussed in 4.3.6) has a bigger influence than we thought on the whole request-response process. Therefore, in the future there have to be a special test that allows HTTP connection testing.

Another interesting conclusion is that when we changed some parameters, it affects other processes. We saw this when EJB requests had to wait very long time because of the EJB container was full. It was resulted in HTTP-connection timeout errors what also remained even when the EJB container had gained room. This behaviour in particular doesn't comply to the model that we used because the request should had been accepted and processed by the EJB container when there was room enough gained. This particular issue need further investigation and/or adjustment of the model.

The EJB-container must be big enough to handle a large number of processes that take long time. This can be done by choosing the right value for the maximum Pool Size parameter. We came to this conclusion because when we sent a few messages with large Timer Condition (see section 4.3.3) we got SOAP errors.

Summarising we can say that we got interesting results that require more research to understand the whole picture better. In the future, every parameter that define the SOAP request-response processing must be tested to define the best configuration or value. This could be done by creating good and adequate test procedures and monitoring the effect of every parameter

6. Recommendations

After a month of research and testing we can admit that, when developing complex distributed systems, there can be always place for optimization that can be discovered during testing. Additionally, there will always be unexpected behaviour on the part of the users, the network, or the supporting infrastructure. Recognizing the existence of these missing optimizations, or performance sinkholes, and identifying the paths to improving them is of the utmost importance throughout the lifecycle of an application.

But still, we believe that there are certain way's to improve application performance, scalability and availability.

6.1. Documentation

One of the problems we found in the beginning of our project is that there was not enough documentation available about the implementation of the Generic AAA server. Because of this lack, future testers can experience the same difficulties. Therefore we recommend very strong to document every change, this would be very helpful for future testers.

6.2. Early performance testing

During our research we found that software performance testing of distributed applications has not been thoroughly investigated so far. The reason for this is, that testing techniques have traditionally been applied at the end of the software process. Also because the most developers relay on the prediction of performance estimation models.

Conversely, the most critical performance faults are often injected very early, because of wrong architectural choices.

Therefore we suggest that software performance, scalability and availability must be tested in the very early stages of development. In the long term and as far as the early evaluation of distributed-application is concerned, we believe that empirical testing may outperform performance estimation models providing more precise and realistic results.

6.3. Client request load

Client request load is a key characteristic for performance prediction. As more concurrent client request arrive at an application server, more contention will be incurred for server threads to process these requests. Consequently there will be longer request queue on the application server, as requests will be waiting for the service. This all leads to an increase in client response time.

As explained in chapter 4.3, we could use only 4 clients for testing. So we could not simulate or test the effect of the large number of clients sending requests at the same time. Based on the results we have received we were not able to see this effect. Therefore we suggest further testing, using a higher number of clients.

We think it's very important to conduct such tests, hence this will give good indication about the fact whether the internal EJB container architecture will scale well to handle significantly increasing requests load.

6.4. Session vs Entity Beans

In J2EE applications, two main application architectures are typically used in server components. First, a session EJB is invoked by a client, and the session bean accesses the database directly using JDBC calls. Alternatively, an EJB entity bean can be introduced to

separate the business data from the business logic in the session bean. The entity bean represents a clean object-oriented programming abstraction. Entity beans are used to encapsulate business data in memory such that a session component (e.g. stateless session bean) can then perform the business logic on behalf of the client by accessing and manipulating business data stored in entity beans. Also, the principle of separation of concerns is observed as the business data is encapsulated in entity beans, and that the implementation of session beans is not littered with database access code.

However, the benefits of the highly modifiable entity bean architecture come at the cost of performance. A performance penalty is introduced by the creation and management costs for the entity bean. In addition, the EJB container has to synchronize the data in the entity bean with the underlying data store, which can incur expensive I/O operations.

During the theoretical part of our research, we discovered that the performance of the entity bean architecture is less than 50% of the performance of the session bean only architecture. We found that this is an interesting fact, but unfortunately, because of the available time, we could not tests this in our research. Therefore we think it is an issue that must be examined in future tests.

Considering that fact that we have been using session beans for testing, we think it is a must to develop a test case that will use entity beans. As stated in previous section, HTTP connections have a limitation of maximum TTL. During the tests we discovered that, when the timer condition higher than TTL, the request always fails.

Knowing that in the future the resources will be accessed using Entity beans, we think that the optimal combination of the higher processing time of the Entity beans and the maximum TTL will have a significant impact on the overall performance and availability. Based on this we think it is highly recommended to look at this issue and conduct tests to see if this is creating a bottleneck.

7. Conclusion

During the project development we tried to implement existing practice for testing Java/J2EE based applications and adopt them to the testing GAAA server.

In course of the project we developed general test model that refers to the current GAAA Toolkits implementation and wrote test program that allows controlling different aspects of AAA server performance. Understanding that this project was the first attempt to make a performance test, we believe that our development will provide a basis for further design of the AAA server tests.

Some of the results we had where similar to what we expected and some were not. We also noticed some unexplainable noise in our charts and unpredictable behaviour of the AAA server. In short, we achieved more then our initial goals but still there are enough questions left for future research. In particular, as the following questions should addressed in the future research:

- Statistical methods for test results processing must be developed
- What will happen if we use a complex policy?
- How is changed the resource utilization, using different requests sequences, e.g., number of requests and mix of valid, not-valid and complex requests?
- What would be the performance of the Generic AAA server if we install it on a server with twice as fast CPU? Would the performance be double?
- What is the effect of database connections on the overall performance?

Another possible testing programme may focus on determining which influence each involved parameter has and what is the best value for those parameters.

Another direction that may be interesting is to model Denial of Service (DoS) attack for the AAA Server what we didn't archived in the framework of this project. While testing we tried to crash the server by simulating DoS attacks. Unfortunately we did not succeed. We now know that the Generic AAA server can process at least 8000 requests at the same, as long the value of the timer is kept 1.

Our current suggestion that that the way the Generic AAA architecture is implemented, has good performance with a high availability, however further testing will help to optimise its performance

8. Acknowledgments

The authors would like to thank Cees de Laat, Yuri Demchenko, Bas van Oudenaarde and Leon Gommans for their guidance and assistance during the research.

Looking back we certainly had a very interesting project and to some extension it was highly collaborative work. We had to adjust our time schedule at some points because of the absence of our mentors. We needed their support to make some adjustments in the AAA servlet code. But finally, when we finalised the test program and started testing, it went very smooth and fast. Also useful advises at the stage of preparation of the final report helped to present our results in the best way.

9. References

Iterative Development: The use of J2EE Best Practices , Prepared By Owen Taylor of The Middleware Company, Precise Software Solutions.

Performance management for the J2EE™ platform, Mika Reivari, Borland.

Testing J2EE Applications, Goran Begik, IBM.

Optimizing for the Java™ Servlet API & Database Access, Sang Shin, Sun.

Performance Analysis and Tuning of the Java™ 2 Platform, Enterprise Edition (J2EE™), Sang Shin, Sun.

J2EE™ Overview, Sang Shin, Sun (www.javapassion.com/j2ee/)

Performance Monitoring of J2EE Application Servers, Adrian Mos and John Murphy (Performance Engineering Laboratory), Dublin City University.

DIAGNOSING J2EE PERFORMANCE PROBLEMS THROUGHOUT THE APPLICATION LIFECYCLE, Mercury Interactive.

Performance Tuning Essentials for J2SE™ and J2EE, Jay Campan and Eric, Muller Borland Software Corporation.

Eclipse:
http://www.onjava.com/pub/a/onjava/2004/02/04/juie.html
https://bugs.eclipse.org/bugs/

Tomcat

http://java.sun.com/j2ee/1.4/docs/relnotes/cliref/hman1/create-http-listener.1.html
http://www.cafesoft.com/products/cams/wa/tomcat5/docs20/index.html

Session beans
http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Session.html
http://www.samspublishing.com/articles/article.asp?p=30109

10. Appendix A.

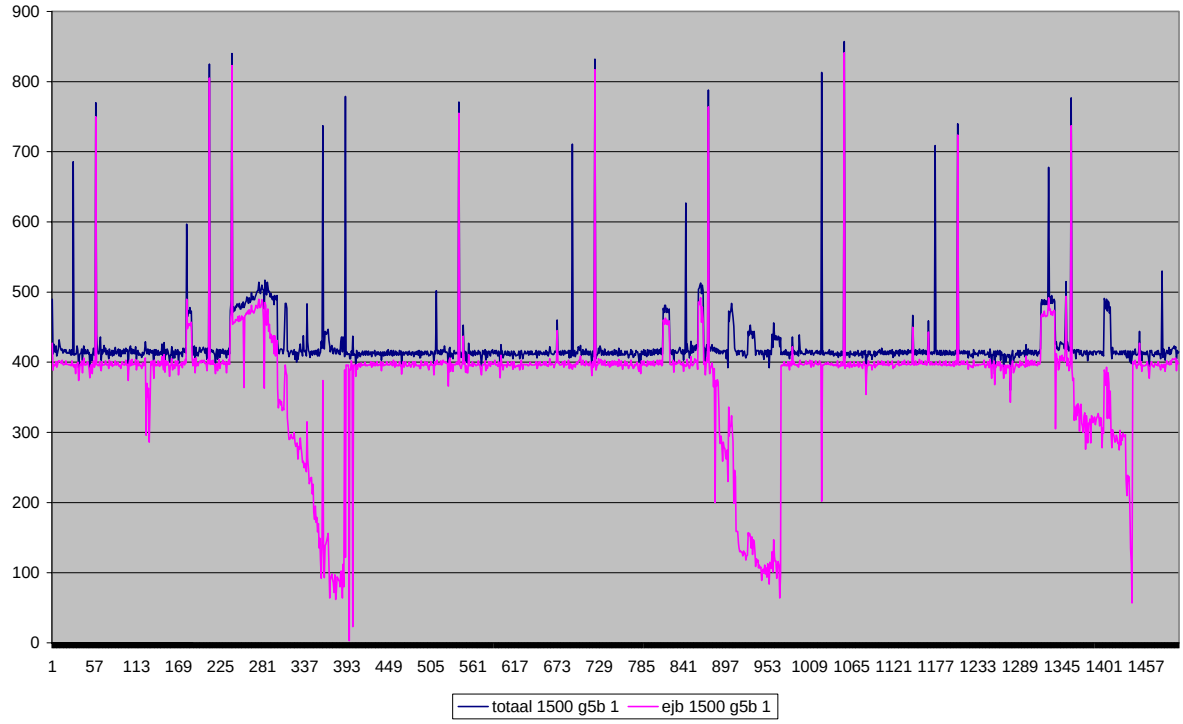

Figure 12 : 1500 requests with standard settings from mac-g5b

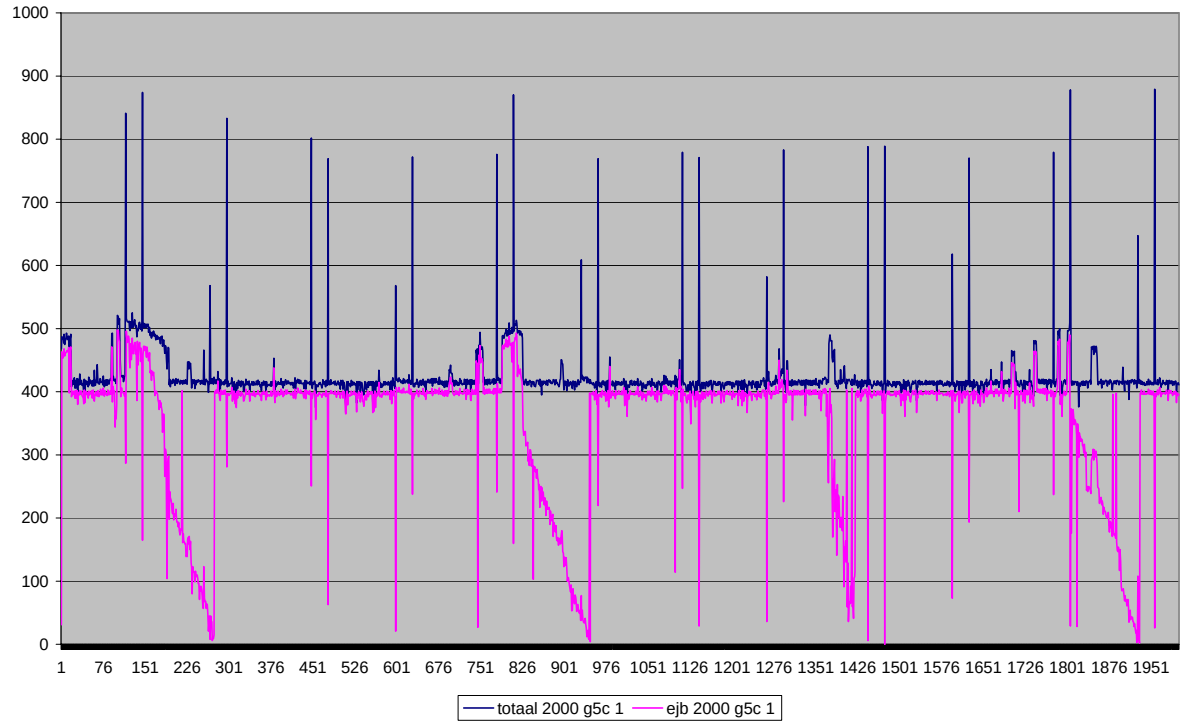

Figure 13 : 2000 requests using standard setting from mac-g5c

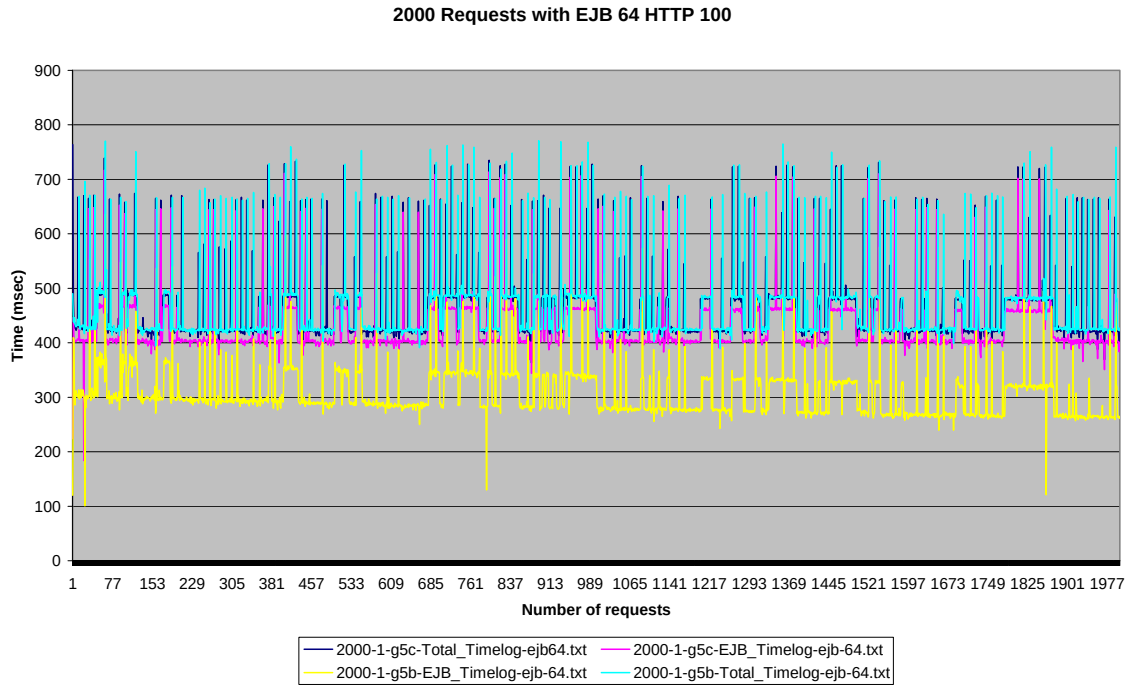

Figure 14 : 2000 Requests with HTTP 100 EJB 64

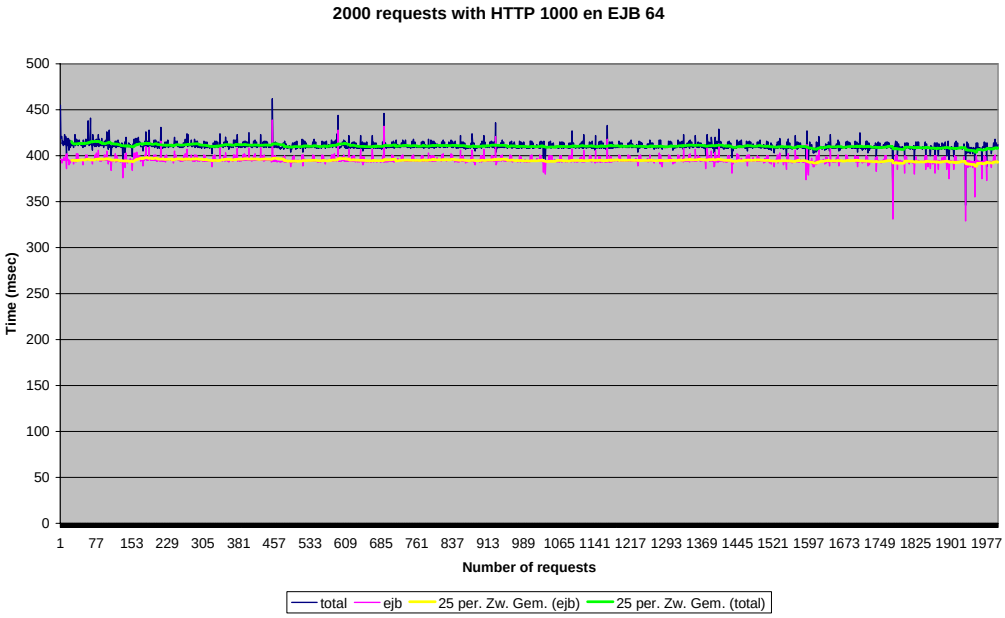

Figure 15 : 2000 requests with HTTP 1000 en EJB 64

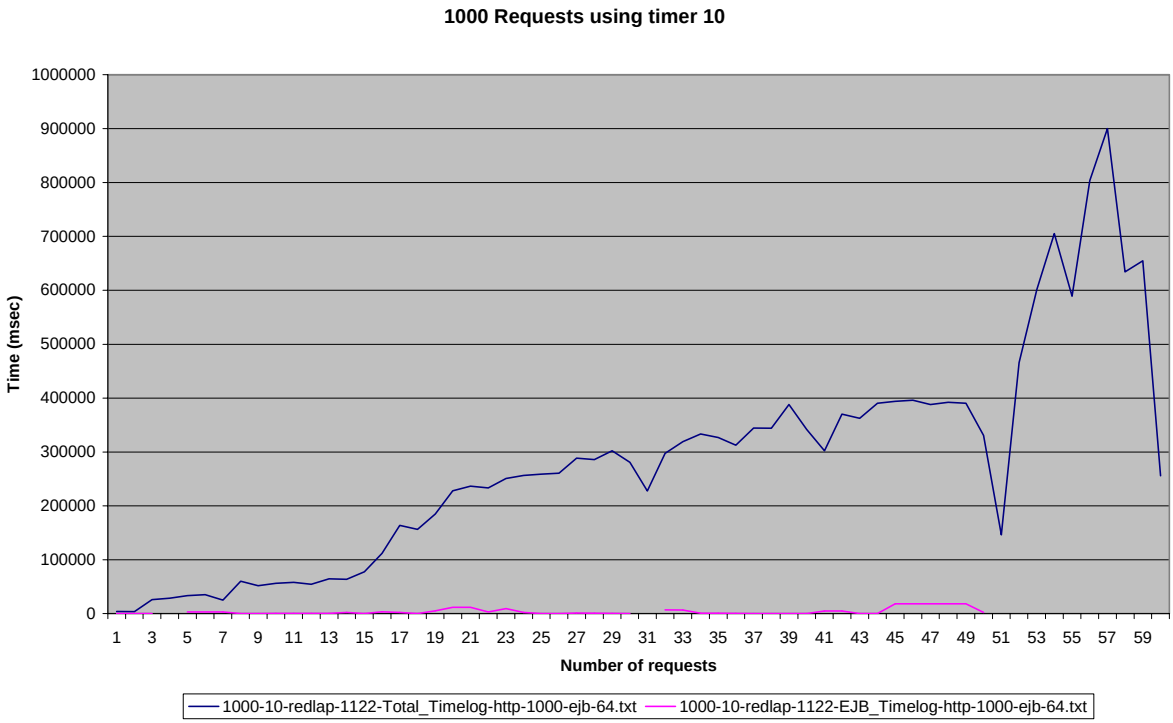

Figure 16 : 1000 requests with timer set to 10

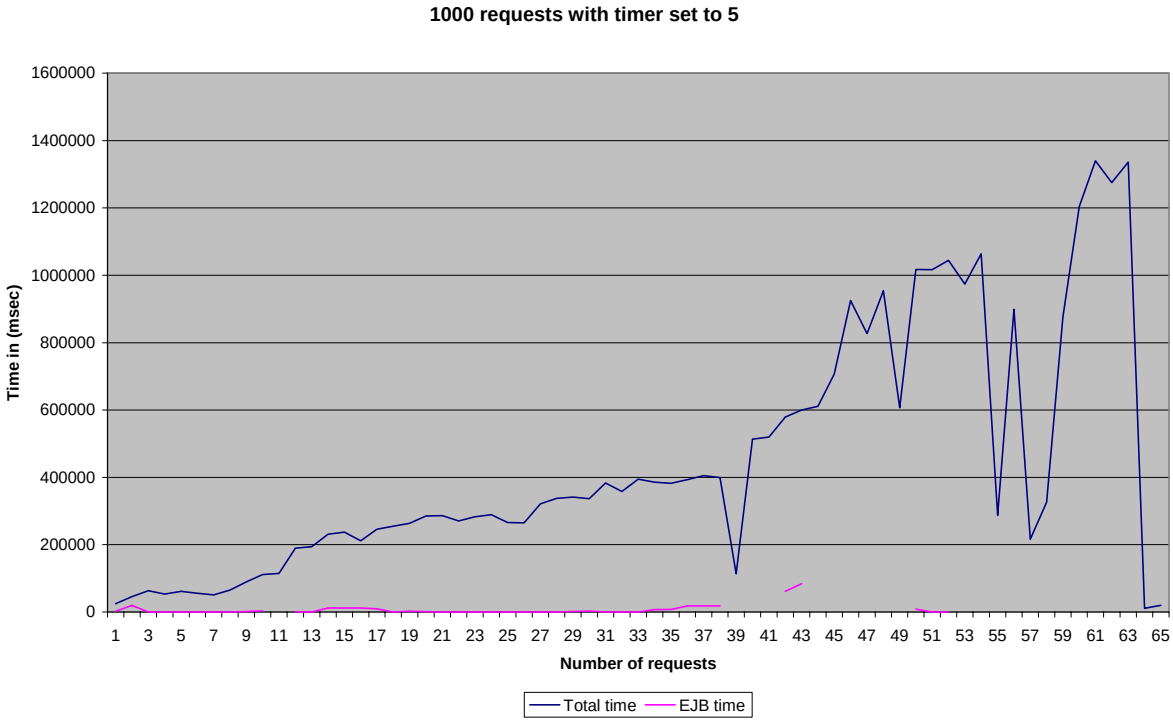

Figure 17 : 1000 requests with timer set to 5

11. Appendix B. Test Program listing

```
/**
 *                  SoapThreadRequest.java
 *                  This file was made for testing how the /AAA/server
 *                  would react on a multithread request and therefore not optimized.
 *                  note that the request are all the same (same .xml file)
 *
 *
 */
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.xml.sax.SAXParseException;
import org.xml.sax.SAXException;
import org.w3c.dom.Document;

import java.io.IOException;
import javax.xml.soap.*;
import java.util.Date;
import java.net.URL;
import java.text.*;
import java.io.*;

class ThreadRequest01 implements Runnable {
        SoapThreadRequest03 Tijd = new SoapThreadRequest03();
        File File4req; //variable needs to be global
        int nrOfReqs; //number given by user (second parameter)

        // F = File4req = File2Send = is tje .xml file that has to be sent
        ThreadRequest01(String F, int intt){
                File4req = new File(F);
                nrOfReqs = intt;
        }

        //The .start() method below (in class SoapThreadRequest) calls the run() method
        public void run(){
                Tijd.Countje++;
                //We used this to know which thread is last. this way there were less access violations
from the threads

                SendReq2Server(File4req, nrOfReqs); //send the request and receives response.
                //try {
                //        Thread.sleep(10);
                //} catch (InterruptedException e) {e.printStackTrace();}
                Tijd.SendResultstoDB();
        }

        //This method does all the work. is starts a Soapconnection and parses the .xml
        public void SendReq2Server(File File2Send, int nrOffReqs){
                final ITimer timer = TimerFactory.newTimer ();    //Java provides an API for this, we did
this in the AAA Servlet
                Document document;

                try{
                        FileOutputStream Total_Timelog, EJB_Timelog; //our log files
                        Total_Timelog = new FileOutputStream ("./Total_Timelog", true); // 'true'
because we want to write to an existing file
                        EJB_Timelog = new FileOutputStream ("./EJB_Timelog", true);
```

```
		document = null;
		DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();
		dfactory.setNamespaceAware(true);
try {
		DocumentBuilder builder = dfactory.newDocumentBuilder();

		document = builder.parse(File2Send);
}
catch (SAXParseException spe) {
		System.out.println("\n** Parsing error"    + ", line " + spe.getLineNumber()
			+ ", uri " + spe.getSystemId());
		System.out.println("   " + spe.getMessage() );
		Exception  x = spe;
		if (spe.getException() != null)
			x = spe.getException();

		x.printStackTrace();
}
catch (SAXException sxe) {
		Exception  x = sxe;
		if (sxe.getException() != null)
			x = sxe.getException();

		x.printStackTrace();
}
catch (ParserConfigurationException pce) {pce.printStackTrace();}
catch (IOException ioe) {ioe.printStackTrace();}

try {
		SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
		SOAPConnection connection = scf.createConnection();
		MessageFactory msgFactory = MessageFactory.newInstance();

		SOAPMessage msg = msgFactory.createMessage();
		SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
		SOAPBody body = envelope.getBody();
		SOAPBodyElement docElement = body.addDocument(document);

		// hier gaan we de SOAPAction header gebruiken voor onze eigen doeleinden
		//MimeHeaders headerz = msg.getMimeHeaders();
		//headerz.setHeader("SOAPAction", "101");

		URL endpoint = new URL("http://195.169.124.61:8080/AAA/server");

		msg.saveChanges();

		//---------- onScreen !! >> ----------------

		System.out.println("\n---------- Request Message ----------\n");
		timer.start (); // start counting
		msg.writeTo(System.out);

		SOAPMessage reply = connection.call(msg, endpoint);
		System.out.println("\n----------------------------\n Response from: "+endpoint);

		System.out.println("\n---------- Reply Message ----------\n");

		reply.writeTo(System.out);
		connection.close();
		timer.stop (); // stop counting
```

```
                        MimeHeaders headers = reply.getMimeHeaders();
                        String[] EJB_Time = headers.getHeader("SOAPAction");

                        //Used to get and set the headers
                        //bron: http://archives.java.sun.com/cgi-bin/wa?A2=ind9810&L=servlet-
interest&F=&S=&P=37271

                        System.out.println("\n\n timer.getDuration = "+timer.getDuration());
                        //like we said. java provides an API for this
                        //we did this in the Servlet

                        new PrintStream(Total_Timelog).println (timer.getDuration()+";");
                        new PrintStream(EJB_Timelog).println (EJB_Time[0]+";");
                        System.out.println(" SOAPAction header = "+EJB_Time[0]);
                        EJB_Timelog.close();
                        Total_Timelog.close();
                        timer.reset();
                }
                catch (Exception ex) {ex.printStackTrace();}
                }
                catch (Exception ex) {}
        }
}

class SoapThreadRequest03 extends Thread{
        static int nrOffReqs;
        int Countje = 0;

        public static void main(String[] args){
                System.out.println("initializing. . .\n");
                System.out.println("please wait. . . . " );
                try{
                        int nrOfthreads = Integer.valueOf(args[1]).intValue();
                        ThreadRequest01 Soaping = new ThreadRequest01(args[0], nrOfthreads );

                        Thread[] tread = new Thread[nrOfthreads];
                        nrOffReqs = nrOfthreads;

                        //Send as many als been given (totalGivenReqs)
                        for(int i=0;i<nrOfthreads;i++){
                                tread[i] = new Thread(Soaping);
                                tread[i].start();   //starts a thread
                                sleep(2000); // wait after every start of a thread
                        }
                }
                catch(Exception e){};
        }
        public void SendResultstoDB(){
        }
        public String easyDateFormat (String format) {
          Date today = new Date();
          SimpleDateFormat formatter = new SimpleDateFormat(format);
          String datenewformat = formatter.format(today);
          return  datenewformat;
        }
}
```

12. Appendix C. AAA Request/Response messages

12.1. The request

```
<AAA:AAARequest xmlns:AAA="http://www.AAA.org/ns/AAA_BoD"
        xmlns:xsi ="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.AAA.org/ns/AAA_BoD
        http://195.169.124.61/SNBrequest01.xsd"
version="0.1" type="SNBpolicy01" >

<TimerCondition>40</TimerCondition>
<TimerAction>10</TimerAction>

</AAA:AAARequest>
```

12.2. The response

```
Response from: http://195.169.124.61:8080/AAA/server
---------- Reply Message ----------
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
  <soap-env:Header/>
  <soap-env:Body>
    <AAA:AAAReply type="BoDReply" version="0.1" xmlns:AAA="http://www.AAASchemas.com/Reply"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <AAA:Answer>
        <AAA:Message>permit</AAA:Message>
      </AAA:Answer>
    </AAA:AAAReply>
  </soap-env:Body>
</soap-env:Envelope>
------------------------------
```