Quality of Service - FFI
IT2901 - Group 7

Bremnes, Jan A. S.
Johanessen, Stig Tore
Kirø, Magnus L.
Nordmoen, Jørgen H.
Støvneng, Ola Martin T.
Tørresen, Håvard

March 7, 2012

Abstract

TODO ...

Contents

List of Figures

List of Tables

1 Project Introduction

This is the midterm report documenting our progress so far in the course IT2901 - Informatikk Prosjektarbeid II. It will give a description of the problem we were presented with, how we have planned the work for the weeks ahead, how we have organized our group and what development methodology we have chosen etc. We will also give a brief discussion about why we have made the decisions we have. For the midterm report we added detailed design description for both client and server and elaborated more on existing parts of the report.

2 Task Description and Requirements

Our task is to provide a Quality of Service[1] (QoS) layer to web services for use in military tactical networks. These networks tend to have severely limited bandwidth, and our QoS-layer must therefore prioritise between different messages, of varying importance, that clients and services want to send. Our software will have to recognize the role of clients, and, together with the service they are trying to communicate with, decide the priority of the message.

2.1 Description

[Description of the task at hand.]

Our assignment is to create a Java application which will function as a middleware layer between web services, and clients trying to use these services. The middleware needs to process SOAP[2] messages, which is the communication protocol for most web services, in order to be able to do its task. On the client side, the middleware needs to process messages and understand SAML[3] in order to deduce the role of the client. This role, together with information about the service the client is trying to communicate with, decides the overall quality of service the messages should receive.

Our software needs to be able to modify the TOS/DiffServ packet header[4] in order for the tactical router[5] to prioritize correctly. Currently NATO has just defined one class, BULK, which is to be used with web services, but this may change in the future and our middleware should handle this upcoming change gracefully.

In addition to this, the middleware needs to be able to retrieve the available bandwidth in the network, which in the real system will be retrieved from the tactical routers. In our testing this information will come from a dummy layer, but how this information is obtained should also be very modular, so that the customer can change how the bandwidth information is obtained later.

With all this information, the role of the client, the relationship between the client and the service, and the available bandwidth, our middleware layer should

[1] Quality of Service refers to several related aspects of telephony and computer networks that allow the transport of traffic with special requirements

[2] A lightweight protocol intended for exchanging structured information in the implementaion of web services in computer networks

[3] Security Assertion Markup Language. See glossary for more info

[4] Type of Service, a field in the IPv4 header, now obsolete and replaced by diffserv. See glossary for more info on IP header and packets

[5] A Multi-topology router used in military networks

be able to prioritize messages. Our product should, as much as possible, use existing web standards, the customer outlined some of their choices and options we have for implementation, like SAML, XACML[6], WS-Security[7] and WSO2 ESB[8]. In addition to this, our middleware needs to work with GlassFish[9], as that is the application server the customer uses.

2.2 Requirements

As the customer wanted all documentation written in English, we decided to use this for all written communication and documentation, in order to keep things consistent.

The way the course is structured in terms of deliveries of reports and documentation also creates a fairly natural implicit sprint period to work off of, and using an agile methodology will help in easily producing and maintaining said reports and documentation. In addition to the reports and documentation, we will try to deliver a prototype to the customer before the final delivery in May.

The customer does not require any prototypes along the way, just a working piece of software by the end of the project, so the deadline we have set for the prototype is self-imposed.

The customer has not given us many strict requirements, but instead they have suggested a few things that we could do. Given this freedom, we decided that we should improve on the base requirements by adding most of the things mentioned in this section.

The following is a list of technology requirements.

- Written in java
- High priority messages must arrive, even at the cost of dropping lower priority messages.
- Use standards where they can be used

  SAML

  Diffserv

  XACML

  WS-Security
- Test thoroughly

  Use NS3[10] for testing
- Extensive documentation
- Use metadata to determine priority
- GlassFish must be supported as the application server

[6] eXtensible Access Control Markup Language
[7] An extension to SOAP to apply security to web services
[8] An Enterprise Service Bus built on top of Apache Synapse
[9] Application server written in Java
[10] NS3 is a network simulator. See glossary for more info

- Must be able to set priority on network layer packets

  Currently there is only one priority class defined by NATO, the BULK class, but this will most likely change in the future, as such our middleware layer needs to be expandable enough to handle this change in the future.

- There are no requirements on resource usage, but we should try to keep it lightweight.

  The customer has only said that we can expect the product to be used on a standard laptop with full Java support

3 Project Management

In this section, we'll take look at how we organized the team, a brief risk assesment, and an evaluation of the work process.

3.1 Team Organization

This section describes in detail how we organize ourselves and how we split roles and tasks among the team members. We have a flat team structure[11] and have shifted our focus accordingly over to team communication.

3.1.1 Team Structure

We already know each other coming into the project so we have chosen a flat organisational structure, with no intervening levels of management, since all decisions within the team will more or less be made by all the members together either way. Relying on the entire group for decisions will both involve and invest everyone in the project and will work well with our already existing group dynamic.

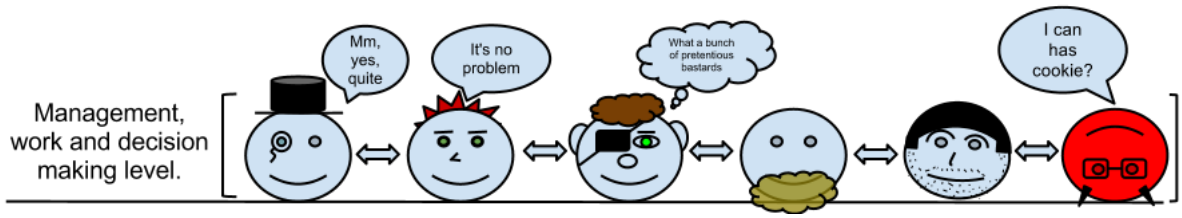


Figure 1: Team Organization chart
It was made during lunch, but the general principle still remains, that the structure is flat.

As the structure shows in the chart, there is no difference in what responsibility level anyone have, or what role one has. The concept of changing roles weekly is good for a learning situation, but very inefficient where knowledge

[11] Flat organization structure is a structure with few or no levels of intervening management. The idea is that well-trained workers will be more productive when they are directly involved with decision making. http://en.wikipedia.org/wiki/Flat_organization

and research are key components in a limited timed project. We anticipate that the time for this project probably won't be sufficient for any role changes, and therefore we have to keep people focused on the task they have been assigned. The efficiency of the current task relies on having the current research fresh in mind. If we were to change the roles every week, the newly assigned person would spend much time getting up to date at the beginning of every week, which in turn wouldn't yield any measurable gains.

Rather than focus on responsibilities within the group, we've chosen to focus on tasks. The task will to some degree still represent areas of responibilities, and since tasks will be spread across several group members, we don't run the risk of a single missing member crippling the entire group. Instead the remaining member(assigned) to a task will be able to pick of the slack. This, together with thorough documentation of a members knowledge, will just about eliminate the problems associated with an absent group member.

Further, the team structure and the distribution of responsibility gives us the chance to define how we want to deal with task and their priority. The work flow that we have makes us prioritise tasks continuously and get the most pressing task done at the correct time. It's similar to a max heap. We put tasks in to the heap, heapify(prioritise tasks) and choose(pop) the maximized task, the task that has the highest priority.

When we choose a task we consider the persons interest, experience and existing knowledge. Most times the tasks fall naturally to one person that has worked with similar tasks earlier in the project. Other times there is more of lottery, where the task has no prerequisites. Often we rely on a persons initiative to take a task or we easily delegate them with a question, "Can someone do that?". Task delegation and sharing the work load has not been a problem so far in the project.

3.1.2 Team communication

We decided that we will work together from 10 to 16, Monday through Thursday every week, with allowed exceptions for lectures and such. Group members can also work in their free time to make up for missed collaboration hours or to just put in some extra work. This means more work than the course requires, but we decided that we want to do it this way so we can either take some time off now and then, or have more time for the exams in May.

We will not be able to have frequent face to face meetings with the customer, but we will have weekly online meetings with them instead, as well as e-mail communication as needed. Since we have seen what happens in projects where there is little to no communication, we decided, in agreement with the costumer, that we at least wanted to have weekly meetings in order to keep a good dialog with the customer, and also give them the opportunity to take part in the development of the project. Because the customer is located in Oslo, we decided that the weekly meetings will be held over Skype.

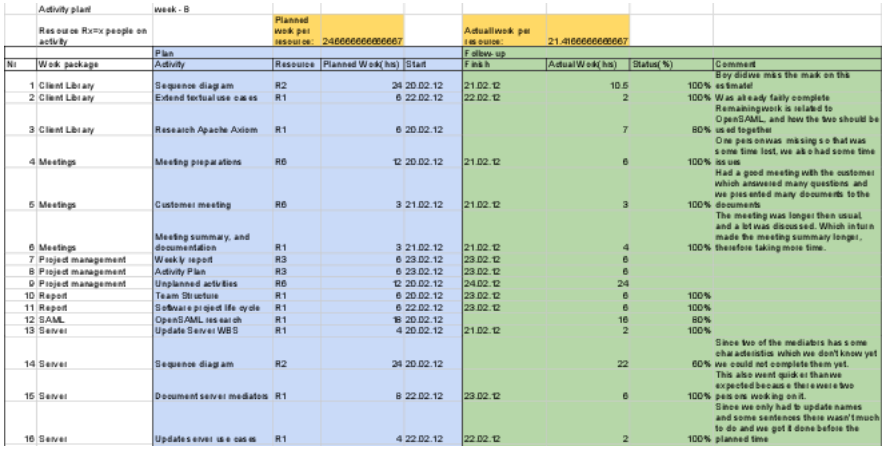
3.2 Risk Assesment

3.3 Process Evaluation

Project management documented: Possible deviations and how they have been handled.

3.4 Progress tracking and Documentation

In the beginning we had a summary every day where we wrote what we were working on and what had to be done. We stopped doing this after we got good activity plans because the daily summaries became unnecessary.



	Activity plan!	week - 8							
	Resource Rx=x people on activity		Planned work per resource:	246666666666667		Actualiwork per resource:	21.4166666666667		
		Plan				Follow- up			
Nr	Work package	Activity	Resource	Planned Work(hs)	Start	Finish	ActualWork(hs)	Status(%)	Comment
1	Client Library	Sequence diagram	R2	24	20.02.12	21.02.12	10.5	100%	Boy did we miss the mark on this estimate!
2	Client Library	Extend textual use cases	R1	6	22.02.12	22.02.12	2	100%	Was already fairly complete
3	Client Library	Research Apache Axiom	R1	6	20.02.12		7	80%	Remainingwork is related to OpenSAML, and how the two should be used together
4	Meetings	Meeting preparations	R6	12	20.02.12	21.02.12	6	100%	One person was missing so that was some time lost, we also had some time issues
5	Meetings	Customer meeting	R6	3	21.02.12	21.02.12	3	100%	Had a good meeting with the customer which answered many questions and we presented many documents to the documents
6	Meetings	Meeting summary, and documentation	R1	3	21.02.12	21.02.12	4	100%	The meeting was longer then usual and a lot was discussed. Which in turn made the meeting summary longer, theiefoe taking more time.
7	Project management	Weekly report	R3	6	23.02.12	23.02.12	6		
8	Project management	Activity Plan	R3	6	23.02.12	23.02.12	6		
9	Project management	Unplanned activities	R6	12	20.02.12	24.02.12	24		
10	Report	Team Structure	R1	6	20.02.12	23.02.12	6	100%	
11	Report	Software project life cycle	R1	6	22.02.12	23.02.12	6	100%	
12	SAML	OpenSAML research	R1	16	20.02.12		16	80%	
13	Server	Update Server WBS	R1	4	20.02.12	21.02.12	2	100%	
14	Server	Sequence diagram	R2	24	20.02.12		22	60%	Since two of the mediatrs has some characteristics which we don't knowv yet we could not complete them yet.
15	Server	Document server mediators	R1	8	22.02.12	23.02.12	6	100%	This also went quicker than we expected because there were two persons working on it.
16	Server	Updates erver use cases	R1	4	22.02.12	22.02.12	2	100%	Since we only had to update names and some sentences these wasn't much to do and we got it done before the planned time

Figure 2: Example Activity plan

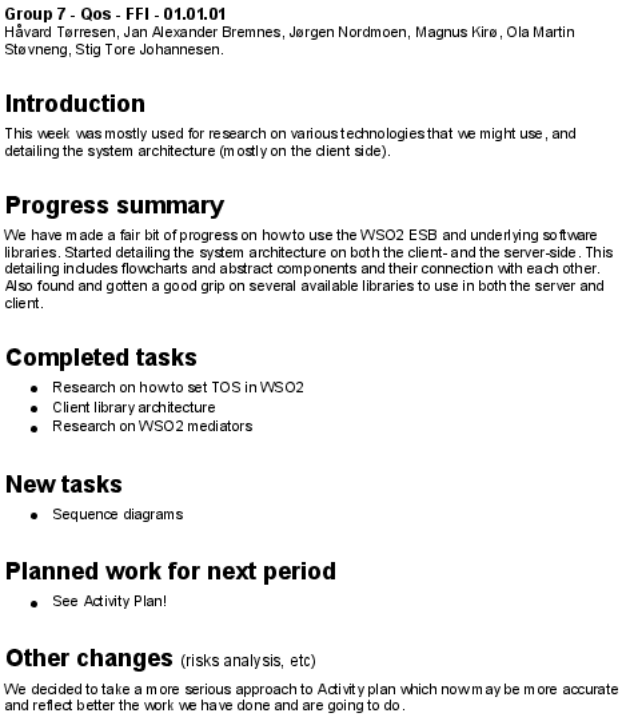
The activity plans(Fig:2) now have the role of our day to day summaries and work progress. We update the activity plan as we go along. This way we have a complete overview of tasks and work hours that are planned this week. As we update the activity plan we have an overview of the work done this week and where we have missed with our time estimation.

As we can se in the weekly report (Fig:3) the staus report ha a standard setup. We created a template early in the project so that we did not have to redo the work later. In the prosess of creating the template we put some thought in to it so that we would get a template that would work throughout the project without further changes.

The activity plan has a template which we clone every week to create the plan for next week. We update it

4 Development Methodology

We did not follow any established development methodology, such as Scrum or XP, as this project required more planning and configuration of existing solutions, than actual coding. We therefore chose a mix of waterfall and agile methods, we discuss these decisions in the sections below. You will also find a list of the tools we chose to work with, and why we decided to use them.



Group 7 - Qos - FFI - 01.01.01
Håvard Tørresen, Jan Alexander Bremnes, Jørgen Nordmoen, Magnus Kirø, Ola Martin Støvneng, Stig Tore Johannesen.

Introduction

This week was mostly used for research on various technologies that we might use, and detailing the system architecture (mostly on the client side).

Progress summary

We have made a fair bit of progress on howto use the WSO2 ESB and underlying software libraries. Started detailing the system architecture on both the client- and the server-side. This detailing includes flowcharts and abstract components and their connection with each other. Also found and gotten a good grip on several available libraries to use in both the server and client.

Completed tasks

- Research on howto set TOS in WSO2
- Client library architecture
- Research on WSO2 mediators

New tasks

- Sequence diagrams

Planned work for next period

- See Activity Plan!

Other changes (risks analysis, etc)

We decided to take a more serious approach to Activity plan which now may be more accurate and reflect better the work we have done and are going to do.

Figure 3: Status report example

Because this is a research project, the customer will act more as an advisor than a customer, and will have more suggestions and advice than demands and requirements. We have been given a clear understanding of what the final product should be, and we have a list of requirements that should be met. Other than that, we are relatively free regarding how we go about solving the problem. Because of this, a single methodology, like Scrum, won't work for us, as it requires us to be in close and frequent contact with the customer, presenting a prototype every other week and continue development based on the customers feedback and demands.

As mentioned, this is a project that requires quite a lot of planning before any programming can be done. This necessitates that we start the development according to a waterfall model in terms of the architecture planning as well as the requirements specification.

As the project progresses we'll be switching to a more agile development method, so as to allow for iterative development and facilitate for any necessary changes that may turn up as code is produced. Agile also allows for the flat organisational structure we have chosen, which we believe will greatly help cooperation within the team.

4.1 Project Organization

[How we organize the project]

We have devided the project tasks in to work packages. These packages are

represented in a work breakdown structure(WBS) (A). The timeplan for the project is represented in a Gantt diagram (Fig.4). The figure is part of our full Gantt diagram. As the full diagram cannot be included nicely in the report we have attached it as an HTML document (B).

Tasks

WBS	Name	Start	Finish	Work	Priority	Complete	Cost	Notes
1	Planning	Jan 18	Jan 20	18d		100%		
2	Work on preliminary report	Jan 23	Feb 3	60d	10	100%		
3	Submission of Preliminary Report	Feb 6	Feb 6					Thu 26 Jan 2012, 14:55 This is a note
4	Architecture planning	Feb 6	Mar 5	126d	9	8%		
5	Work on midterm report	Feb 6	Mar 9	150d	7	0%		
6	**Work on prototype client**	Mar 8	Apr 16	84d 2h				
6.1	Open SAML	Mar 8	Apr 16	27d 4h		0%		
6.2	**Client library**	Mar 8	Apr 16	56d 6h				
6.2.1	Metadata interpreter	Mar 8	Mar 22	10d 3h		0%		
6.2.2	Prioritizer	Mar 22	Apr 9	13d 3h		0%		
6.2.3	Tactical router communication	Apr 9	Apr 16	5d 7h		0%		
6.2.4	**Interface**	Mar 8	Apr 16	28d				
6.2.4.1	Client integration manual	Mar 27	Apr 16	14d 6h		0%		
6.2.4.2	API	Mar 8	Mar 27	13d 2h		0%		
7	**Work on prototype server**	Mar 8	Apr 16	131d 5h				
7.1	ESB	Mar 8	Mar 23	12d		0%		
7.2	Identity server	Mar 20	Apr 3	11d		0%		
7.3	Glassfish	Mar 29	Apr 16	12d 4h		0%		
7.4	**SAML mediator**	Mar 8	Apr 16	27d 4h				
7.4.1	?	Mar 8	Apr 16	27d 4h		0%		
7.5	**QoS mediator**	Mar 8	Apr 16	41d 3h				
7.5.1	Metadata interpreter	Apr 2	Apr 16	10d 3h		0%		
7.5.2	Prioritizer	Mar 8	Apr 6	22d		0%		
7.5.3	Tactical router communicator	Mar 28	Apr 9	9d		0%		
7.6	**WSO2 network layer**	Mar 8	Apr 16	27d 2h				
7.6.1	?	Mar 8	Apr 16	27d 2h		0%		
8	Creation of test suite	Mar 13	Apr 6	38d		1%		
9	Testing of prototype	Apr 9	Apr 16	11d		0%		
10	Work on final report	Mar 12	Apr 16	152d 2h		0%		
11	Submission of Alpha	Mar 9	Mar 9					
12	Submission of midterm report	Mar 9	Mar 9					
13	Submission of Beta	Apr 16	Apr 16					
14	Submission of final report draft	Apr 16	Apr 16					
15	Bug fix, polishing, wrapping things up, buffer	Apr 17	May 25	28d 1h		0%		
16	Deadline	May 25	May 25					

Figure 4: Part of our Gantt diagram
Se full diagram in attachments: (B)

4.2 Software project life cycle

For our project life cycle we chose agile. Originally we started out with the intention of using Scrum[12] and Scrum only. That idea was quickly scrapped as we found out that our task was very research heavy. This made us rethink our approach to the development cycle and turn in the direction of agile software development[13]. Early in the project we expected that we could begin coding and prototyping before too long. This proved to be wrong as there was a lot of research to be done. Scrum was originally a tactic to improve product flexibility and production speed. This works very well in software development when you already know what you are supposed to do and the major part of the task is to implement the required functionality. When the functionality has to be designed and researched extensively scrum becomes unsuitable. With the agile method there is elements that suits us better then others. "Individuals and Interaction"

[12]Scrum is a form of agile project management [Wikipedia: Scrum]

[13]Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams [Wikipedia: Agile Software Development]

and "Customer Collaboration" are two important elements that we use. The full description of the agile method can be found in the Agile Manifesto[14]. Individuals and interactions are strongly connected with the organization of our team (3.1). The flat team structure force us to have a good dialog among the group members. This increase the team members interaction and strengthens the team communication. The strengthened communication promotes the individuals of the group and the team members confidence, which in turn increases the total productivity of the group. The frequent interactions with the customer are also a part of our adaption to the agile development method. Customer Collaboration is the aspect of the group contacting the customer and keeping a good dialog with them. This is to make sure that we produce a product that the customer wants. To achieve this part of the agile manifesto we have meetings with the customer every week and have frequent email correspondence to iron out the bumps of our product. The frequent communication with the customer helps us to create a more precise and consistent system with better documentation. The main part of the communication with the customer is for the benefit of the project and constant improvement. The constant improvement and iterative work flow is a central part of the agile method.

4.3 System Technology

We intend to do a test driven development in order to achieve high quality code. This will give us something to test while we are working, and it will also give us a great way to tell if some new piece of code gets in the way of previously written code. For this purpose we will use JUnit[15] as the testing framework. We will also be doing periodical code reviews approximately every two weeks of development, synchronized with a code/feature freeze where we make sure everything works. As the customer wanted extensive testing of the middleware, we agreed to do testing on the network emulator NS3, as we have someone in the group already familiar with it. The advantage of using NS3 will be extensive testing, but also a great deal of empirical and verifiable data, which the customer also can use to evaluate the product.

We will use Git[16] and GitHub[17] to handle our file repository, although Google Docs will be used for easy sharing and collaboration of schedules, meeting minutes, and reports. Even though the course set us up to use Subversion, we decided against this as Git gives us more options to develop code which will not greatly affect other parts of the code base before we decide to integrate it. To this extent we have decided that as much as possible we should take advantage of Git's built in support for 'branches'. The argument for using Google Docs is that we have the possibility of editing a document together and easy sharing of documents. Delivered reports will be created with LaTeX[18], which we prefer over standard word processors. Our GitHub repository is open to the public, and the software will be released as open source, most likely under the

[14] Agile Manifesto, the key elements of the agile software development method. [AgileManifesto.org]

[15] A testing framework for the Java programming language

[16] A free and open source, distributed version control system

[17] A web-based hosting service for software development projects that use the Git version control system

[18] A document preparation system for the TeXtypesetting program

Apache License.[19]

Each of us is free to choose his own IDE for programming. Because we are using Git, there should be no problem in using the IDE of our choice, and this gives us the added advantage that each person can use the tool which he is most comfortable with. We will stick to the standard Java Coding Conventions.

Since we were so free to choose which tools we wanted to use we decided that this list should be quite lightweight. However the list compiled should be an indication of what is needed for the project. Some of the tools were chosen by us as is and other were demanded by needs of other components. All tools used can be upgraded, downgraded or dropped during the course of this project. The final report will contain the official list as such this list is not in any way final. Our final report will also contain a list with supported tools tested with the final product.

- Git version 1.7.x
- Java version 1.6.x
- Free choice of IDE
- JUnit version 4.x
- NS3 unknown at present time
- WSO2 ESB 4.0.3

5 Prestudy

This project is one that requires quite a lot of prestudy before we can begin coding or even designing the architecture. Since the customer wanted us to implement existing technologies, such as Glassfish, WSO2, SAML etc. we need to spend some time researching those technologies to figure out what to use, and how to use it. The following sections will describe the overall architecture of how we, at this point in time, imagine our system will be like.

5.1 Server side Architecture

The server side architecture consists of several components, the WSO2 ESB, the WSO2 Identity Server, the Tactical Router and the GlassFish server. All of these components are already available, so what we will have to make is mediators[20] in the ESB.

Before the client can request a web service it has to have an identification. To get an ID-token it has to contact the Identity Server using the ESB as a proxy[21] (Fig.6-1). Then the client can request a web service from the ESB. Several things will then happen in the ESB. First the request message is sent to the SAML mediator (Fig.6-2), this mediator contacts the Identity Server to validate the clients ID-token (Fig.6-3). If the token is validated and the client

[19] http://www.apache.org/licenses/LICENSE-2.0.html

[20] A component in WSO2 ESB which van be used to work on incoming or outgoing messages that passes through the ESB

[21] enter proxy definition here

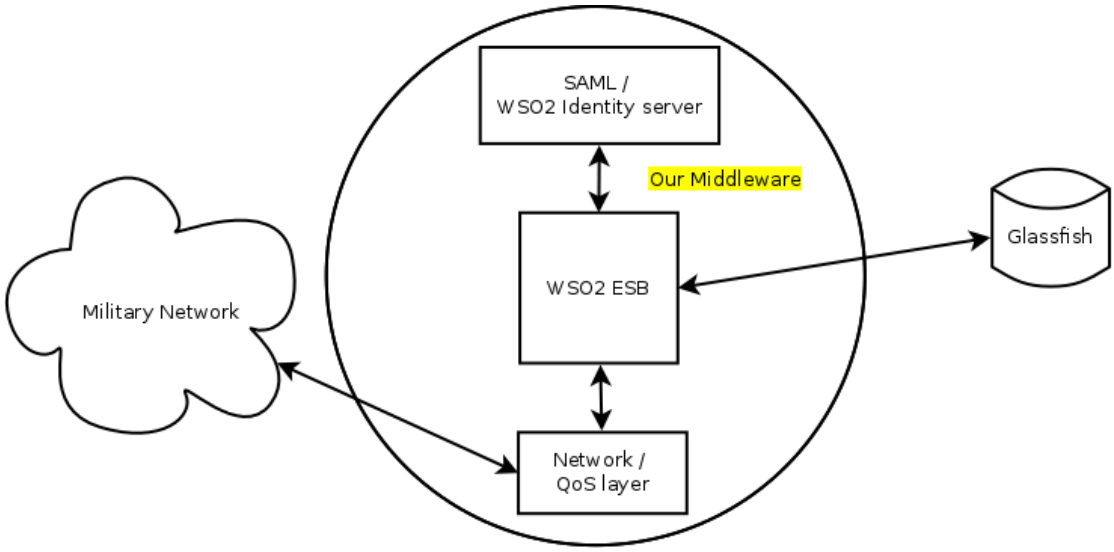


Figure 5: Birds eye view of the overall architecture.

is supposed to have access to the requested service, the message is passed on to the GlassFish proxies (Fig.6-4), otherwise it is dropped. The ESB acting as a proxy will then send the request along to the requested service on the GlassFish server (Fig.6-5).

When the request is received at the service, it will probably start sending some data to the client. This is also done through the ESB. First the message is sent to the QoS mediator (Fig.6-6). This mediator will first look at the role, or identity, of the client and the service requested, and use this information to assign a priority to the connection. Then the Tactical Router is contacted for bandwidth information (Fig.6-7), which is used together with the priority to determine whether the message should be sent right away or held back until some higher priority message is finished sending.

Either in the QoS mediator, in the ESB's network layer, or after that, the Diffserv (ToS) field of the IP header will have to be set (Fig.6-8) before the message is sent to the client (Fig.6-9). This field is used by the routers in the network to prioritize packet sending. This step is quite important to the whole procedure as this is one of the few requirements the customer has given us, as such this step can not be dropped from the final product.

5.2 Client side Architecture

The client-side architecture will be composed of altered (already existing) client software, the OpenSAML library as well as our client library implementation.

Before the client library can ask for the data the client needs to get a SAML authentication token from the identity server (Fig.7-1). The communication here will most likely be handled by our library, but the SAML packages will be created and analyzed by the OpenSAML library.

The client library then sends the request from the client to the server (Fig.7-

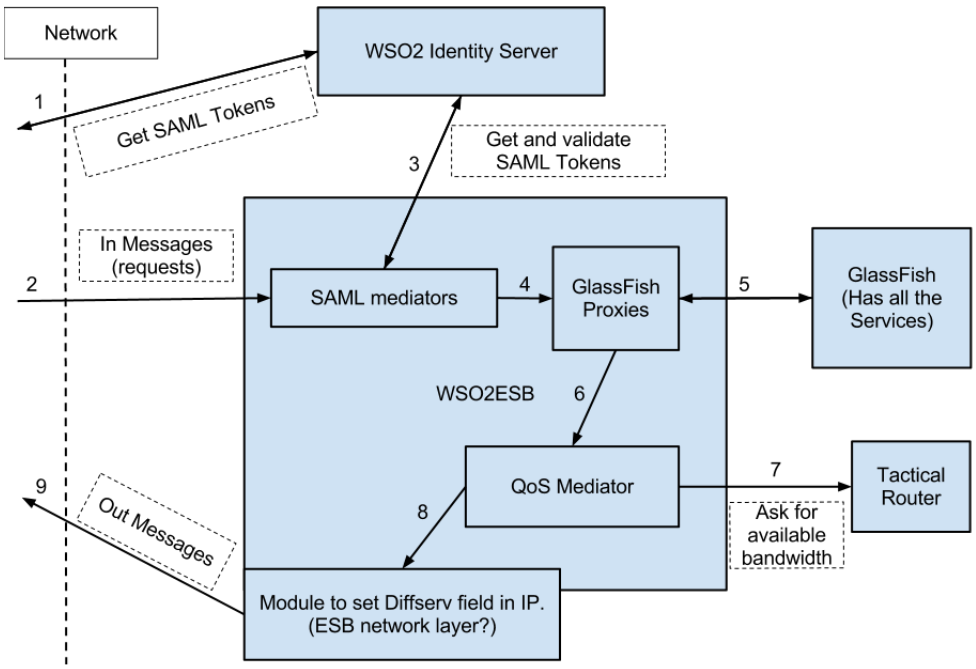


Figure 6: The Server side Architecture

2), appending the SAML token to the package as well as adding some metadata in the SOAP header related to the client role and setting the TOS field of the package to a default value.

The reply from the server is examined by our client library for the metadata the server has embedded in the SOAP header, relevant metadata is stored for future communication and the package is passed to the client application (Fig.7-3).

When new communication is initiated after this first connection is made the client should, if everything went as expected, have the necessary information to prioritize new messages. This means that the client can now take an informed decision about how it should prioritize messages, but in order to do this to the best of it's abilities it also has to take into consideration available bandwidth (Fig.7-4).

5.3 Alternative solutions

[Solutions that we have thought about but discarded.]

The customer also gave us a paper[**?**] which described a previous project they had worked on which tried to solve something quite similar to what we were tasked with. The paper described a system which were used in conjunction with Tactical routers(tactical router) to retrieve bandwidth information and to control sending of messages into this network. As the customer explained this work was not something we could directly copy as the project had not used a lot of web standards and had focused more on the tactical routers as opposed

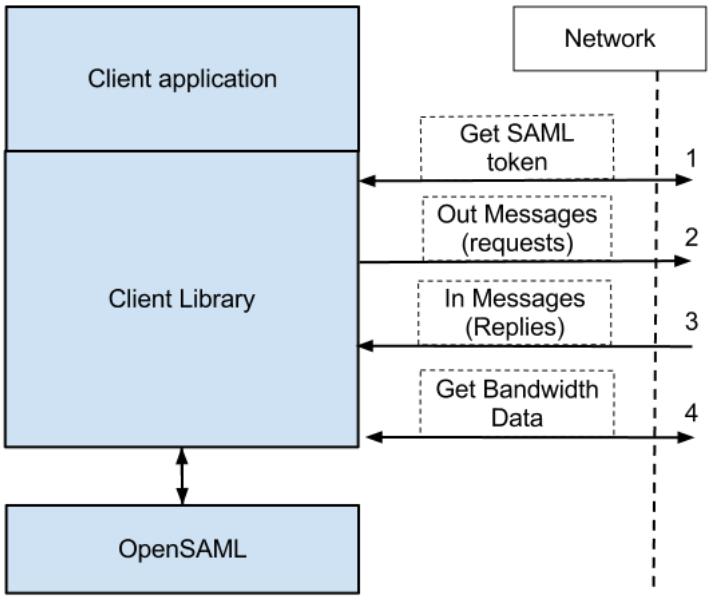


Figure 7: The Client side Architecture

to web services. What we could take out of it however was how they throttled messages. The paper contained five methods which we could easily implement and use their result as an indication of what methods we should use to throttle or hold back messages.

One architecture, which our customer suggested for the project, was to have a proxy in between nodes and creating a custom QoS layer which would sit in front of both the client and the services. This layer would then communicate with a SAML server for authentication, and would have to do all the message prioritization based on the same criteria as our architecture. There are several points about this architecture which would make it a good fit for us. Since the QoS layer would be identical on both client and server side it would mean less work, and more code that could be shared among components, but this freedom comes with some downsides. The first and most glaring problem encountered would be that services on the server would have to be altered to be able to communicate with this front end. Even though we were free to choose architecture ourselves, the client expressed a wish that we would not choose this model because the customer wants to use COTS[22] services which would not be compatible with the new front end.

Even though the above mentioned architecture is not the best fit for us we wanted to take some aspects of the architecture further. Since clients can easily be altered, the above mentioned solution is not applicable for server side, the solution could however be used for the clients. Having a proxy on the client side could be quite good, but because of the work involved and probable time constraints we chose not to go with this solution. On the server side however a front end is not the best solution for us. What we instead are looking into, is to

[22]Commercially available Off-The-Shelf

use an ESB(WSO2 ESB) which would be configured together with the services and work as a proxy. Because many ESBs have integrated SAML processing we could easily take advantage of such facilities along with custom message processing, with which we would then extend the ESB to support our needs. The clients would have to point to the ESB, but this should both be trivial to do and the customer has expressed their agreement that this is satisfactory. We could eventually expand the functionality with service discovery, which then would be a good solution to the problem.

So far we have outlined major alternative architectures which could be alternatives to our project, but there are also alternatives within our proposed solution. One such alternative is not to use a premade ESB, but rather build one ourselves. This solution was thoroughly investigated, but was eventually turned down because of the massive amount of work that had to be done, the quality of an already made ESB is much higher than we could ever achieve during this project, and lastly, the open source tools available to implement the functionality needed for SAML was not very well documented, and would take considerable time to get familiar with.

On the client side we also have the choice of having either a HTTP proxy or writing our own custom library. Both have some advantages and disadvantages, a proxy would be better for integration with client programs, but creating this proxy or configuring and customizing an already existing solution is not trivial. On the same note, creating a library for use in client programs is easier, but this would mean that client programs would need to be altered to be usable with our middleware, which isn't that desirable. We chose to go down the road of least resistance, as we see it currently we would have do quite a lot of research into proxies which could in the worst case scenario result in just wasted time as far as our product goes. A client library would from our perspective be easier as we would have more control, the overall design should be easier and we know that with this sort of library we can integrate OpenSAML which is a huge advantage.

6 Design

6.1 Client Side

[The client side design.]

This chapter will introduce the design and architecture of the client side of our system. Section 6.1.1 will introduce the different components that make up the entire client, and includes a description of the different components that make up the client library. Next, section 6.1.2 will describe the use cases, and section 6.1.3 will take care of the data flow, followed by a detailed architecture description 6.1.4. Finally, section 6.1.5 will go through the sequence diagrams.

6.1.1 Introduction

The client architecture consists of the following components: The client application, the client library and the monitoring service. Additionally, the library makes use of some external components to do some of its work.

Component description:

Client application:
The user-controlled applications that utilize web services. These must be modified to send all communication through the client library in order to get the prioritization it should have.

Client library:
This component will handle all communication with the service providers, as well as authenticating users and prioritizing their messages, based on who they are, and what their current role is. The authorization will involve a component from the server side of the project, the identity server, which returns a token if the client is authorized. Client applications connect through a simple interface to provide credentials and data.

Monitoring service (MS):
An external service, most likely residing on a tactical router(TR), that exposes the bandwidth of the active links on the TR it resides on. this information can be used by the library to help decide if there is enough free capacity on the link to send more data, or if something must be done in order to maintain more connections over that link. For our implementation we will most likely make extensive simplifying assumptions about this service since this is still in the research and development stage and therefore a proper implementation of it is unavailable.

External libraries:

OpenSAML:
This component will be used to parse and manipulate XML data in the form of SOAP and SAML. These are fairly extensive and complex data structures so a easy to use external library is essential here.

Apache HTTPComponents:
A lightweight component for easily setting up and using HTTP connections. While not strictly necessary this component will allow us to connect and communicate across networks far more easily than built in java components.

6.1.2 Use Cases

Title: Accept client info
Actors: Client software, Client Library Interface
Main:

1. Client software connects to the library interface
2. Client delivers its credentials
3. Credentials are passed from interface to sequencer.
4. Credentials are sanity checked by sanity checker and passes.
5. Credentials are passed from sequencer to Token manger, then to credential storage.

6. Credentials are stored
7. Buffer for previous tokens is flushed

Extension:

4a. Credentials are obviously invalid

5a. Return error

Title: Accept data to be sent
Actors: Client software, Client Library Interface
Main:

1. Client delivers data it wishes to send
2. Data is passed to the sequencer.
3. Sequencer creates DataObject

Extension:

none

Precondition: Client has established connection to the Library interface and delivered its credentials.
See: accept client info

Title: Fetch Bandwidth info
Actors: client library, Monitoring Service(MS) (probably on tactical router(TR))
Main:

1. Establish connection to MS
2. Read MS's bandwidth info for the route between it and the server.
3. Set bandwidth info for this connection in the DataObject.

Extension:

3a. Bandwidth info not in MS

4a. Connection is local, set bandwidth info to a default for local network.

Precondition: Client has delivered library both credentials and data to be sent (with a destination).
See: Accept client credentials and Accept data to be sent

Title: Connect to server
Actors: Client Library, Server
Main:

1. Connection manager connects to the server
2. Set priority on socket based on SAML-token and related metadata

Extension:

1a. Unable to connect to server

2a. Return error

Precondition: DataObject has been created, and contains both bandwidth info and a token
See: Accept client credentials, Accept data to be sent and Fetch bandwidth info

Title: Get SAML token
Actors: Client library, Server
Main:

1. Client library sends client credentials to server
2. Server verifies the credentials
3. Server returns SAML-token
4. Token is parsed into Token object
5. Token object is put into DataObject.

Extension:

- 2a. Client credentials not valid
- 3a. Server returns error
- 4a. Client library throws error

Precondition: Client has given library credentials and data to send, and a SAML token for the destination doesn't already exist. Connection to server established.
See: Accept client credentials, Accept data to be sent, Fetch bandwidth info Connect to server

Title: Transaction towards server
Actors: Client lib, server, client
Main:

1. MessageHandler sends buffered data to server
2. Server returns reply to data.
3. The ReturnObject in the DataObject gets the data from the server.
4. MessageHandler send data to sequencer.
5. Sequencer sends data to interface (QosClient)
6. Client fires a data received event to all listeners.

Extension:

2a. Server unavailable, reply doesn't arrive within timeout, etc.

3a. Throw error.

Precondition: Data to send exists, SAML token is in cache, connection to server active.
See: Accept client credentials, Accept data to be sent, Fetch bandwidth info, Connect to server Get SAML token

6.1.3 Textual Data Flow

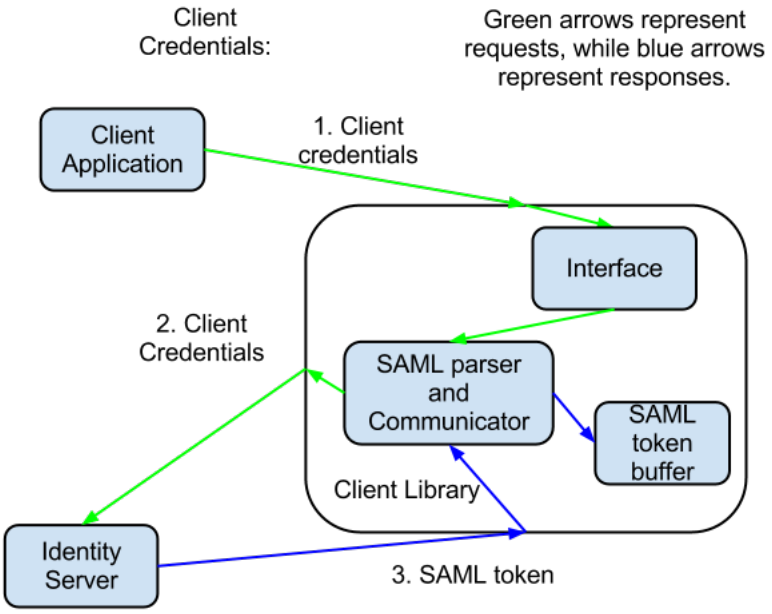


Figure 8: Client Data Flow

Client credentials

1. From client application
2. To client library via API/interface
3. Via SAML to ID-server
4. Back to client library as valid SAML-token
5. Stored in library until client App changes credentials

Bandwidth/link data

1. Generated by all Tactical Routers
2. Delivered to clients and servers on request

Client data

1. Generated by client application
2. Sent to client library via API/interface
3. Buffered in library
4. Sent from library to server
5. Answer returned to client through library

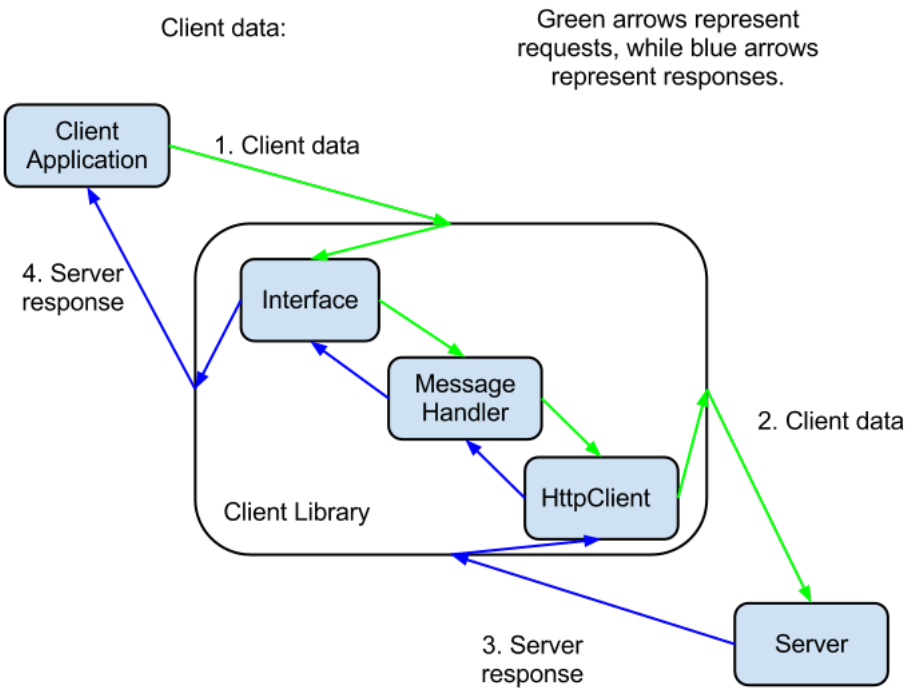


Figure 9: Client Credentials Flow

6.1.4 Architecture

Interface
Known in the class diagram as "QosClient", responsible for providing a clean and easy to use interface for the clients.

Sequencer
The central piece of the client library. Responsible for keeping a record of all other modules in the system and communicate between them as well as making sure everything happens in the right order.

Sanity checker
This module is simply there to do some easy verification of data that comes from the client to make sure that it isn't faulty in any obvious way. (e.g: Data that isn't xml, or credentials that are empty)

Token Manager
This provides a nice and clean interface for the sequencer to store credentials and fetch tokens for data transmissions.

Saml Communicator

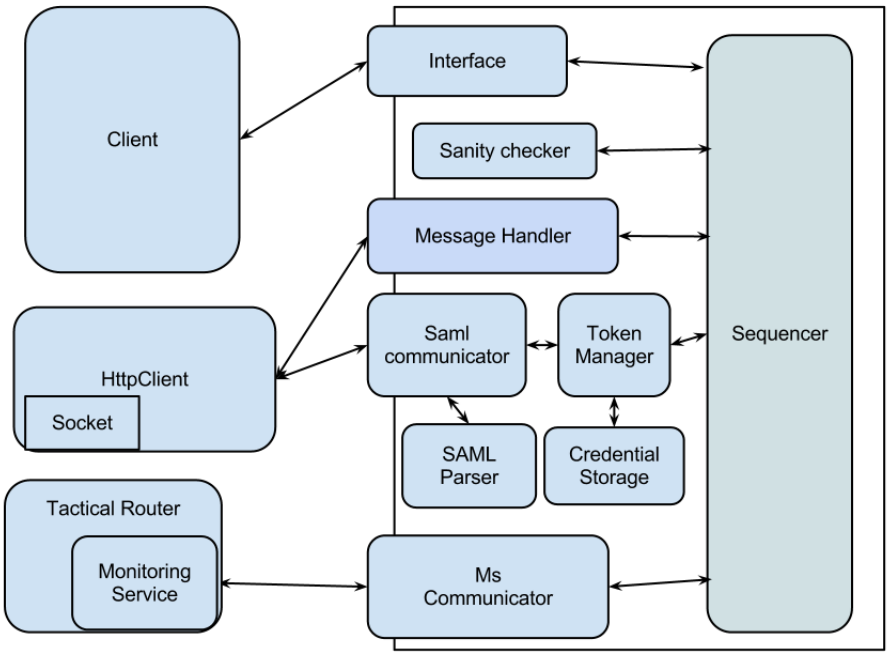


Figure 10: Detailed Client Architecture

This module will take care of the communication between the client library and the identity server.

Saml Parser
This takes the reply from the identity server and parses it into a token object so that it can be easily used and stored.

Credential Storage
Responsible for storing token objects as well as user supplied credentials. Also makes sure that no token objects are returned if they are invalid or expired.

MsCommunicator
Responsible for communicating with the monitoring service to get information about the bandwidth available to a given destination, as well as supplying some identifying information for the route used.

6.1.5 Sequence Diagrams

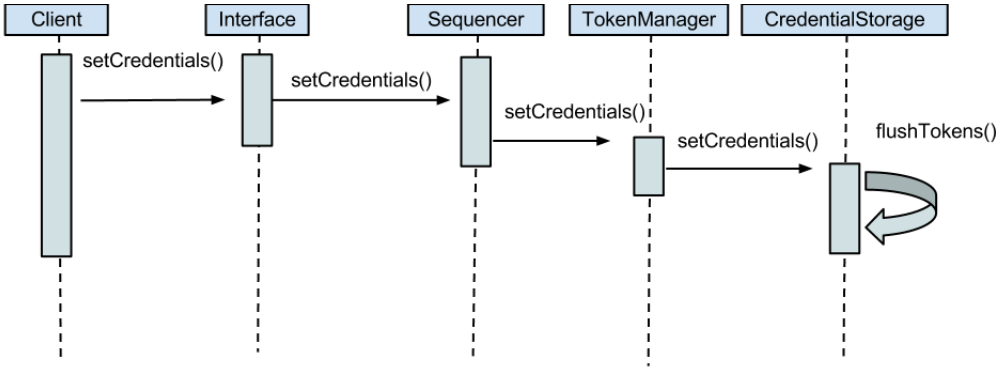


Figure 11: Accept client info

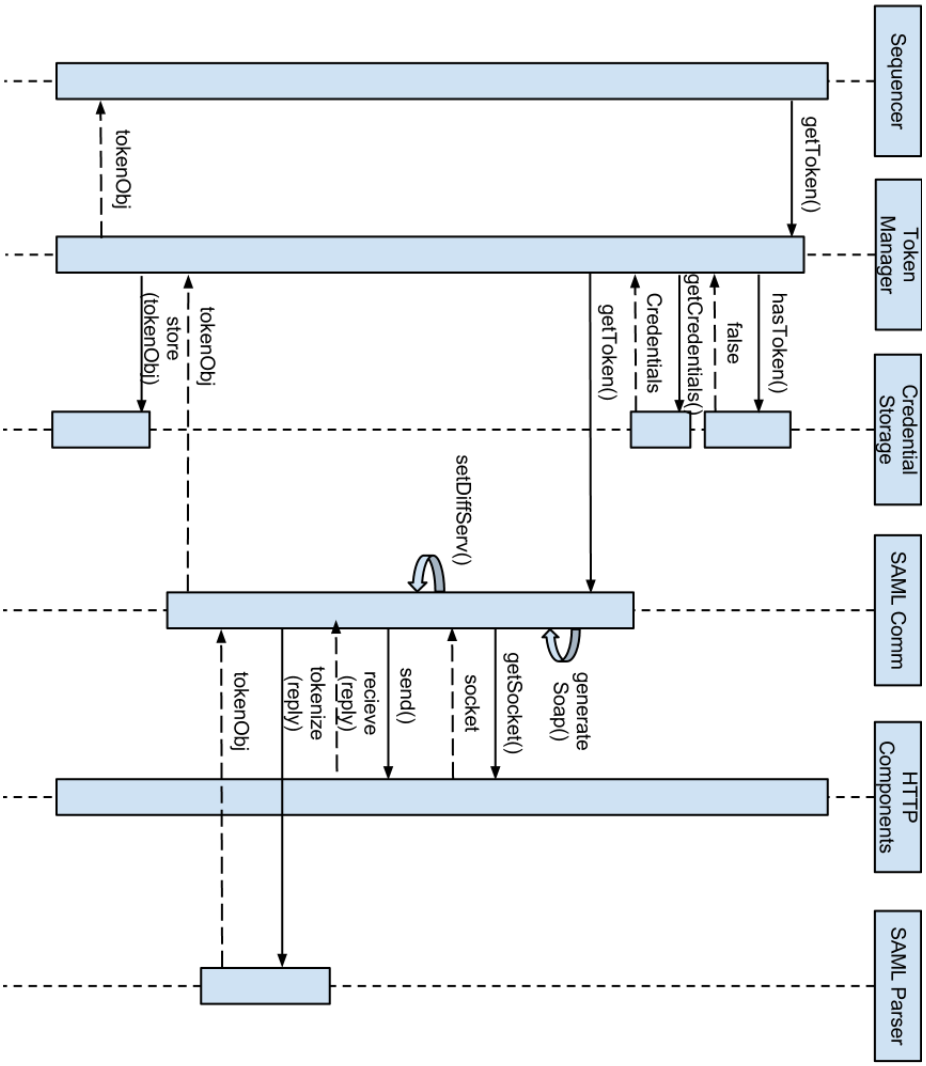


Figure 12: Getting non-stored token

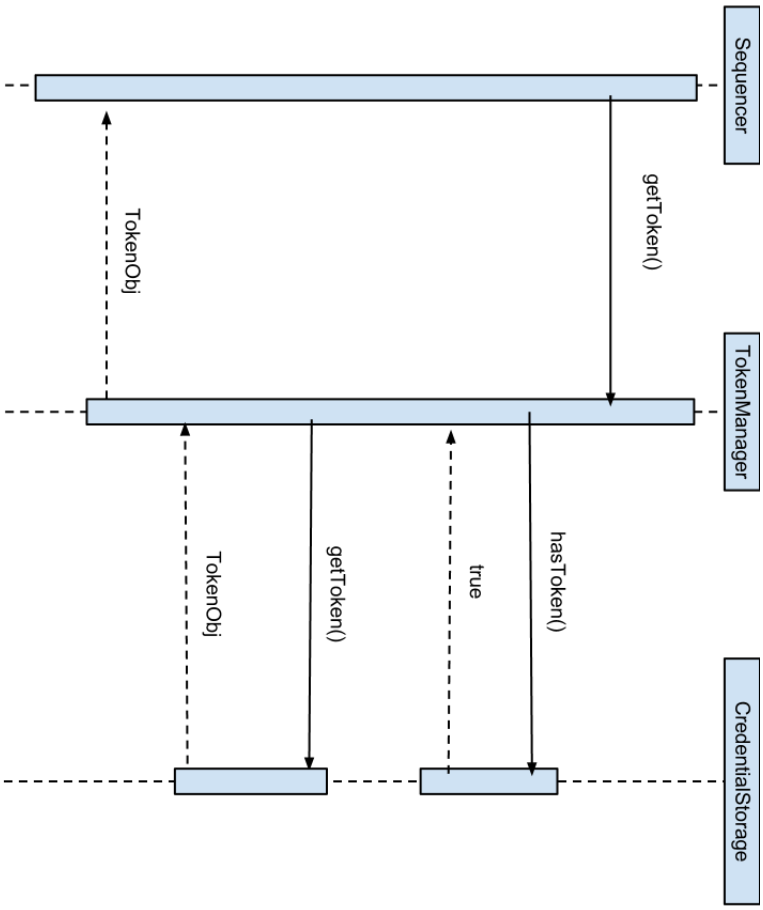


Figure 13: Getting stored token

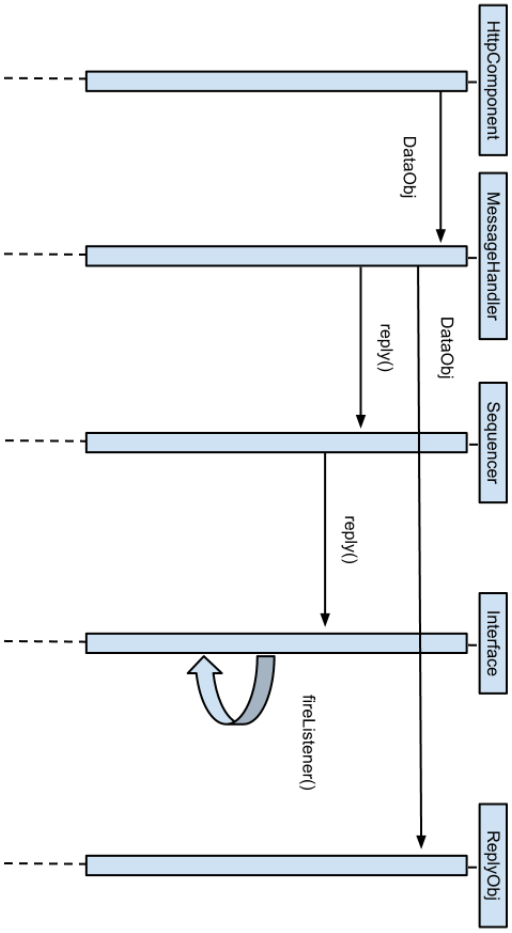


Figure 14: Receive reply

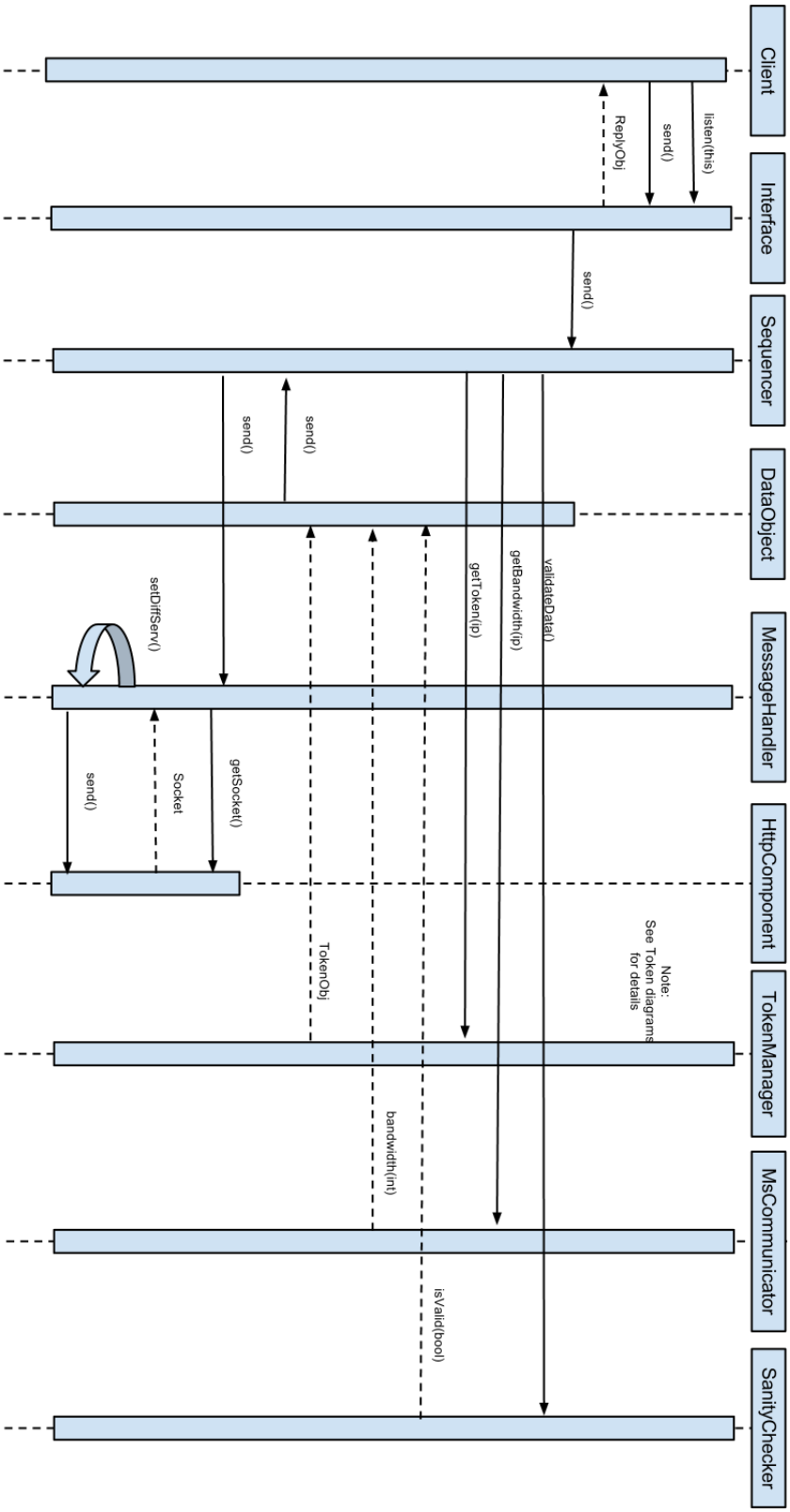


Figure 15: Send data

6.2 Server Side

[The server side design specifics.]

In this chapter we are going to introduce the design, configuration and modification that we are going to do on the server side. In section 6.2.1 we are going to introduce the framework that we have built upon and what we are going to do with it. Next follows use cases, 6.2.2. Section 6.2.3 will go into more detail about what the framework consist off. The section will also guide you through the basic processing units which is used in the framework. The next section, 6.2.4, contains the dataflow through the server side which will help you get a good overview of our thoughts about the design. 6.2.5 goes into detail in describing our custom components in the framework and together with the dataflow should give you a good understanding of the whole server side. Together with the section 6.2.6 you should get good overview even down to some of the details. Section 6.2.7 will give you the details about how we have configured the framework, it will not contain description of how we have set variables during testing, but using this description should make it possible to get the framework up and going. The last section, 6.2.8 will depict how we have modified the framework to be able to meet our requirements, with this section you should be able to tell what modifications were needed and how we have altered those pieces.

6.2.1 Introduction

The server side architecture consists of several components, the WSO2 ESB, the WSO2 Identity Server (IS), the Monitoring Service (MS) and the GlassFish server. The GlassFish server is not necessary to modify, the MS is something we must assume exist in the network and in the IS we only have to configure it to work with the ESB. The ESB is what we have to modify, configure and extend to meet the requirements set.

The ESB will be used to implement QoS for the web services. To do this it will have to communicate with the IS, and the MS in addition to the clients and the services. The ESB must be configured to work as a proxy for the services on the GlassFish server and the IS. It will also be configured to use certain mediation sequences for incoming requests and outgoing responses. The extensions to the ESB consists mainly of custom mediators used in the mediation sequences. These mediators will have the tasks of determining priority of messages, contacting the MS for bandwidth data, and enforcing the priority. There will also be made modifications to the ESB source code to allow for setting the diffserv field in the IP header.

6.2.2 Use Cases

This section will outline the use cases that we have thought of in relation to the server side. With the help of these you should get a rough idea of what we want the server side to be able to do.

Title: Request for SAML authentication
Actors: Client, Enterprise service bus(ESB) and Identity server(IS)
Main

1. Client sends a SOAP message to ESB containing credentials

2. ESB mediates the message to the IS
3. IS fetches SAML Token
4. IS sends the message to ESB
5. ESB adds metadata to SOAP message
6. ESB sends the message back to the client

Extensions:

3a. Invalid credentials

4a. IS returns error message

6a. ESB sends error message to client

Precondition:

- There exists a connection between the client and the ESB

Title: Request mediation
Actors: Client, ESB, GlassFish
Main

1. Client sends SOAP message with SAML Token to ESB proxy
2. ESB authenticates SAML Token(See Authentication use case)
3. ESB removes SAML metadata from message
4. ESB adds metadata to message context.
5. ESB sends message to GlassFish endpoint

Extensions:

2a. SAML Token is invalid

2b. ESB sends error message to client

Precondition:

- Client is connected to ESB

Title: Response mediation
Actors: Client, ESB, GlassFish
Main

1. GlassFish sends message to ESB
2. ESB sets priority metadata in message context and SOAP header.
3. ESB retrieves bandwidth information (See Monitoring Service communication use case)
4. ESB prioritizes message (See Prioritize message use case)
5. ESB sends message to Client

Extensions:
Precondition:

- Request mediation

Title: Monitoring Service communication
Actors: Monitoring Service(MS), ESB
Main

1. ESB requests bandwidth information from MS to a specific address
2. MS returns bottleneck bandwidth to the ESB, as well as the address of the last Tactical Router before the endpoint.

Extensions:

1a. ESB specifies an invalid address

2a. MS returns no information

2b. Address is in the same sub net as the ESB

Precondition:

- Response mediation

Title: Prioritize messages
Actors: ESB
Main

1. ESB acquires QoS information (by magick)
2. ESB adds QoS information to the SOAP header of the message
3. ESB sets diffserv field in IP header

Extensions:
Precondition:

- Response mediation
- Monitoring Service communication

Title: Authenticate SAML token
Actors: ESB, IS
Main

1. ESB sends SAML token to IS
2. IS authenticates the token
3. IS sends verification to ESB

Extensions:

2a. SAML token is not valid.

3a. IS sends error to ESB

Precondition:

- Request Mediation

6.2.3 Description of ESB concepts

In this section we will shortly describe some important concepts of the ESB and message mediation.

A mediator is the basic processing unit in Apache Synapse. Each message going through the ESB gets mediated through a sequence of mediators which can be configured through either XML or WSO2's graphical user interface. As long as the mediator inherit from a Synapse interface any custom mediator can be used in the same manner as the built in mediators. To control the flow of messages through the ESB there are two paths that can be controlled, the "in sequence" and the "out sequence", which can also be configured to only apply for certain endpoints.

The ESB is built around the notion of a message context, this object contains all the information regarding the message and the context around it. In the message context we can add properties, we can manipulate the message it self and we can manipulate the sending streams of the message. All the properties added during the receiving of a message is also added to the outgoing message which we can use to our advantage.

Each mediator in the sequence get access to the message context of the incoming or the outgoing message and can thus manipulate the context to its liking. When the mediator is done with the work it is supposed to do it either call the next mediator sending it the possibly altered message context or returning true to indicate that is done.

6.2.4 Dataflow

This section describes the data flow through the ESB with the help of two diagrams. As a bonus, these diagrams show the general architecture of the server side very well.

Service Request :
To follow this flow follow the green arrows in the figure 16. The ESB receives a request message from a Client, it is then authenticated or if not an error is sent back to the client. If it is not authenticated the flow stops, otherwise the message is sent to the SAML mediator, and then to the send mediator which sends it to the service endpoint on the GlassFish server, and the flow is over.

Service Response:
To follow this flow follow the blue arrows in the figure 16. The ESB receives a response message from the Service, it is then sent through a sequence of mediators, first the Metadata mediator, and then the Store mediator. The Store mediator stores the message in the Prioritized Message Store. The message is stored in the message store until the Sampling Message Processor picks it out before sending it on to another sequence of mediators. First in the sequence is the MS mediator, then the Throttle Mediator and finally the Send mediator. The send mediator sends the message back to the client and the flow is completed.

SAML Authentication Request:
The ESB receives a request (from a Client) directed at the Identity Server (IS),

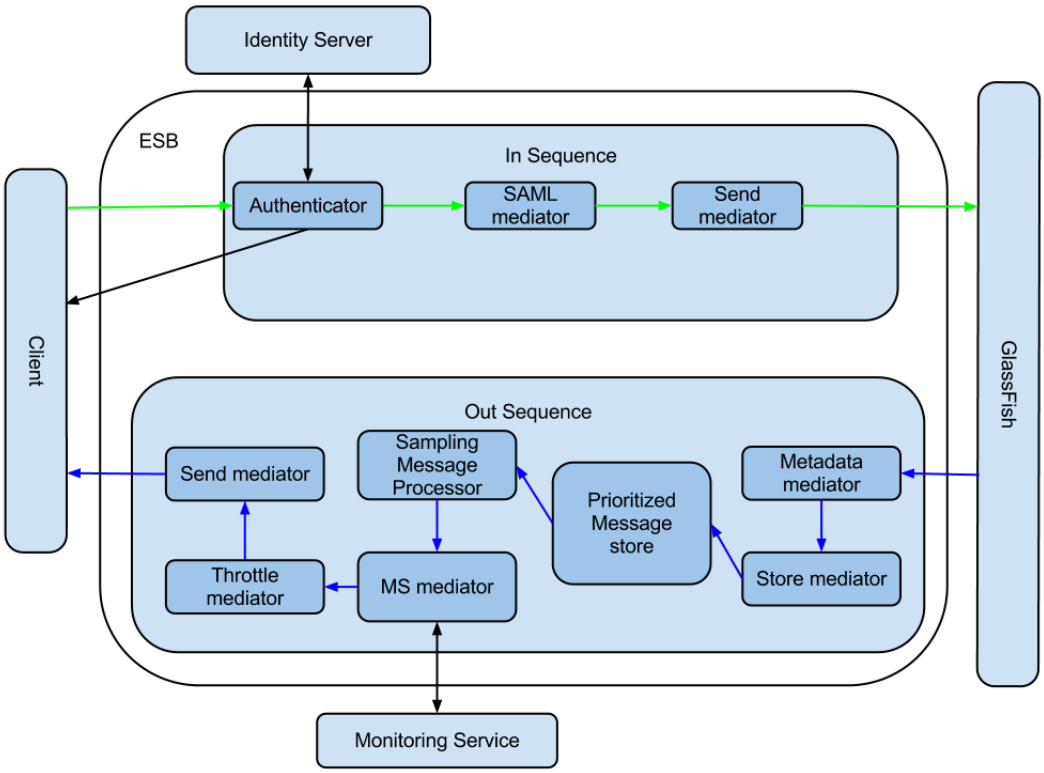


Figure 16: Server Data Flow

the ESB relays this message to the IS. The ESB receives the response from the IS, and sends it through a mediator sequence consisting of the Priority mediator and the Send mediator. The send mediator sends the response to the Client.

6.2.5 Extensions to the ESB

This section will contain a textual description of all the mediators used in the ESB. First we will describe all the custom mediators and extensions we make to the ESB, and then a short description of the built-in mediators we will use.

Custom mediators:

SAML mediator:
This mediator retrieves the user role from the SAML authentication and set this as a property in the message context. The service is retrieved from the endpoint reference and set as another property.

SendBack mediator:
This mediator sends a SAML authentication error message back to the client. After the error is sent to the client the message is forwarded to the Drop mediator. We are not quite sure if this mediator is needed or not, it is difficult to

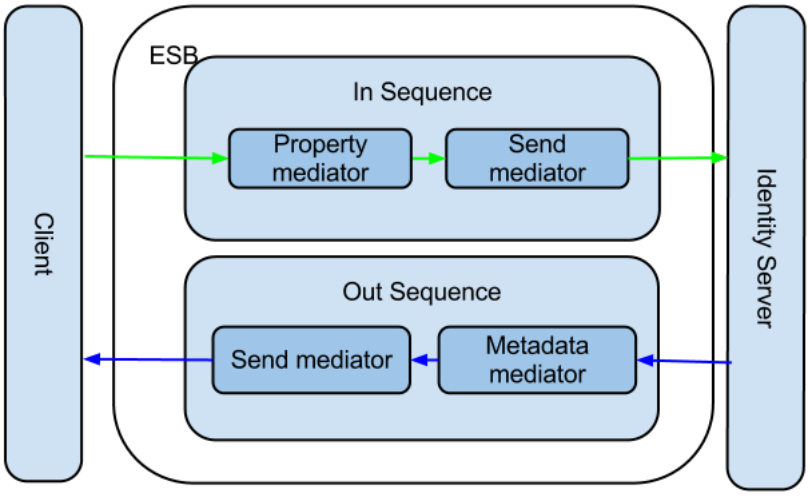


Figure 17: SAML Authentication Flow

determine this before having at least some other parts of the system ready for testing.

Metadata mediator:
This mediator retrieves the client role and service properties from the message context. These properties are then used along with a persistent registry to infer a priority for the message, and what the diffserv field in the IP header should be set as. The priority and diffserv values are then set as new properties in the message context. The diffserv property in the message context will be used in the synapse core to set the diffserv field before sending the message (See 6.2.8).

Prioritized Message store:
This is not a mediator, but it is an important part of the response mediation sequence. This is a message store that stores messages in a priority queue. The queue is ordered by mainly the priority property of the message context, and secondly by the time when added. When retrieving messages from this store the message on the top of the queue is returned. This ensures that high priority messages is processed further before lower priority messages.

MS mediator:
This mediator retrieves the IP address of the receiving client from the endpoint reference in the message context. It sends this IP address to the Monitoring Service and gets the IP address of the last Tactical Router on the path to the client, as well as the limiting bandwidth on the path. The mediator then set this information as properties in the message context before sending the message to the next mediator.

Throttle mediator:
This mediator is used to ensure that high priority messages are sent first by disrupting already sending messages, and it tries to ensure that the network is

not being overflowed by this server by holding back messages. To determine what to disrupt and what to hold back, and for how long several properties are used, the priority of the message, the available bandwidth, the IP address of the client side Tactical Router and the real time demand of the request. In order to do this the mediator must keep a list of sending messages and where those messages are going. This mediator does not have a companion sequence diagram as of the midterm report. The reason behind this is currently a lack of possibility to test the mediator, this will however be remedied when we start to implement the mediators. Once we are more sure of what this mediator will be capable of we will make a sequence diagram for it.

Built in Mediators:

Drop mediator:
This is a built in mediator that drops the message preventing further processing.

Send mediator:
This is a built in mediator that sends the message to an endpoint (the requested service).

Store mediator:
This is a build in mediator that stores the message context in a message store, here this is the Prioritized Message store.

Sampling Message Processor:
This also, is not a mediator. It is a built in class that takes messages out of the Prioritized Message Store at a defined interval. And then sends them to a mediator sequence, here starting with MS mediator.

6.2.6 Sequence Diagrams

This section contains some sequence diagrams which you can use to get a more in dept look into the code and methods used in the mediators above.

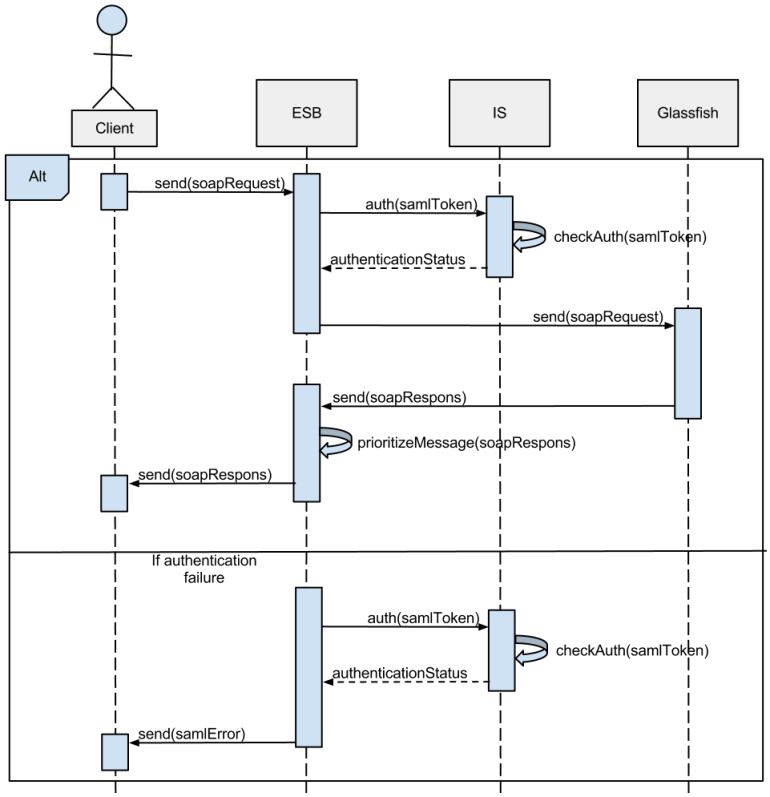


Figure 18: System-level sequence diagram
This high level diagram shows how the client communicates with web services through the ESB.

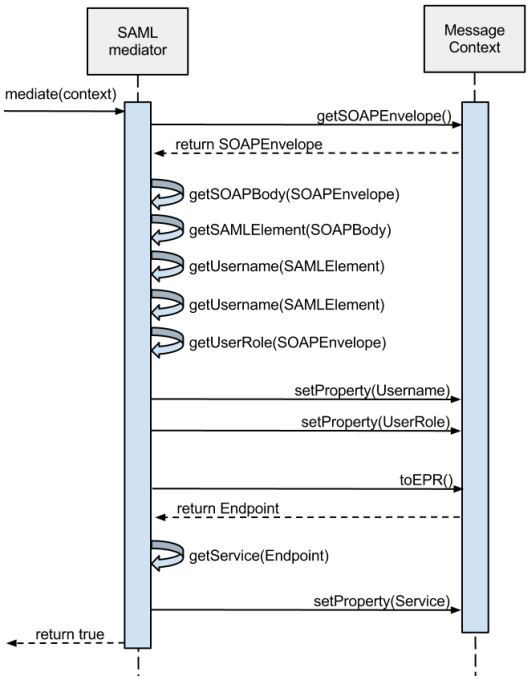


Figure 19: SAML mediator sequence diagram
This diagram describes how the SAML mediator will get data from the message, and set it in the message context so it can be used later in the response sequence

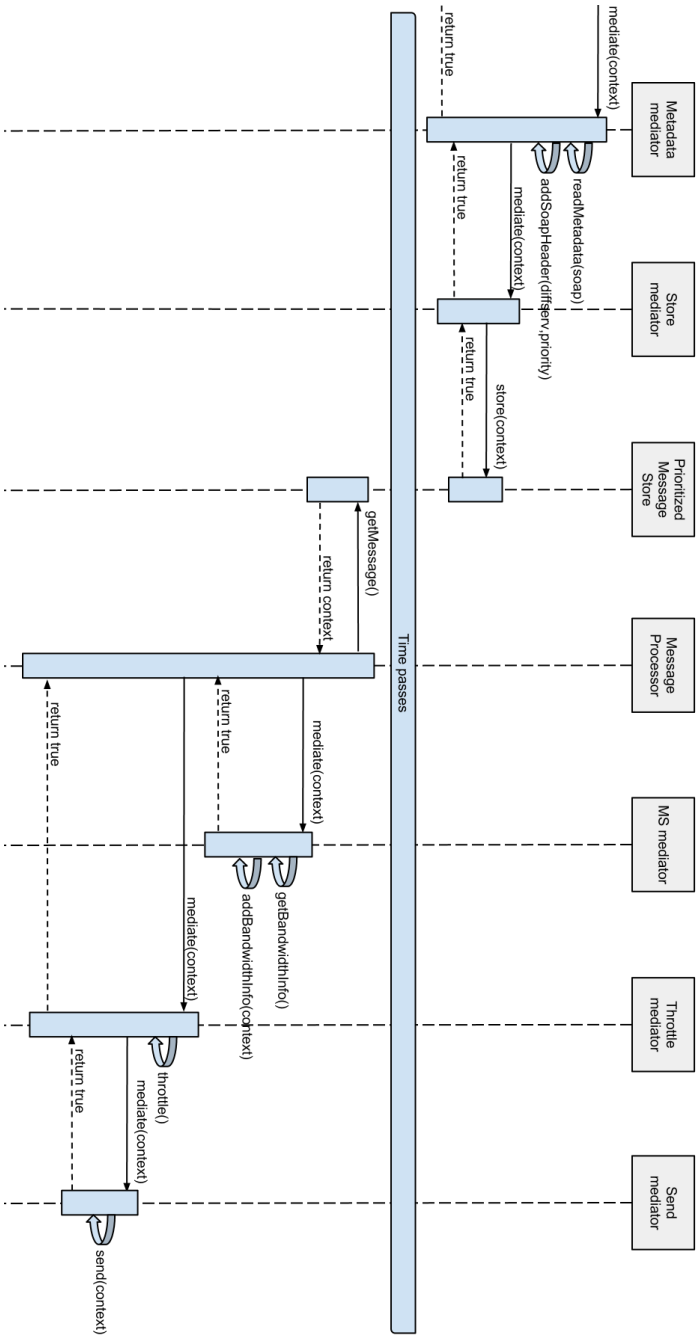


Figure 20: Response sequence sequence diagram
This diagram describes in some detail how a response message from the web service to the Client is passed through the ESB.

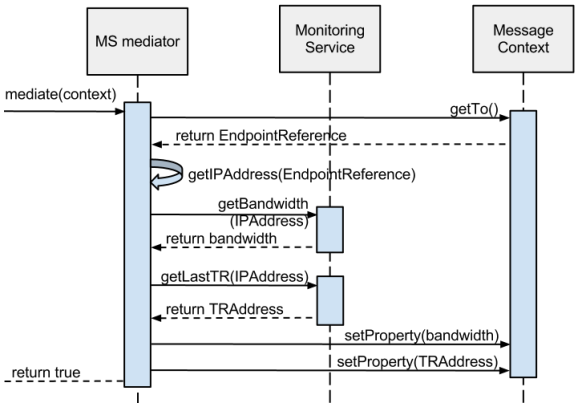


Figure 21: Metadata mediator sequence
This diagram describes how the Metadata mediator retrieves previously stored properties from the message context, determines a priority for the message, and sets priority and diffserv properties in the message context

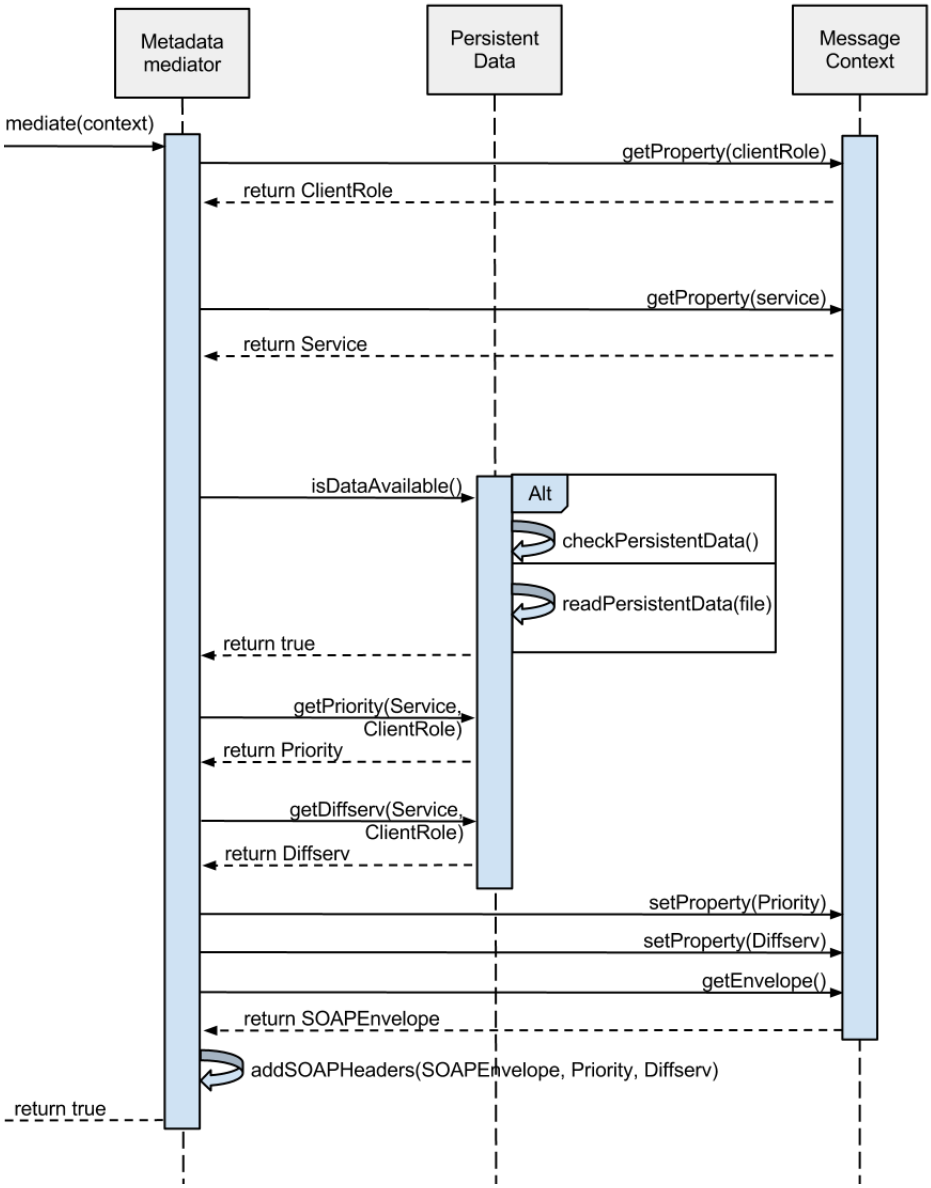


Figure 22: MS mediator sequence diagram
This diagram describes the MS mediator contacting the Monitoring Service for bandwidth information

6.2.7 Configuration of the ESB

This section will be about how the ESB is to be configured.

6.2.8 Modification of the ESB

The WSO2 ESB source code will have to be modified to allow for setting the diffserv field in the IP-header of packets sent. The idea is that we will set a property, DiffServ, in the message context, and let the httpsender get this property and set it on the socket used for sending.

The source code for all the dependencies of WSO2 ESB is included in its source code. As such we only altered files in this source. This made it easier for us to build and create a runnable instance of WSO2's ESB. To also try and support future versions of Apache Synapse and WSO2 we have also applied the changes to the latest version of the underlying libraries.

In Apache Synapse we have altered the way it sends responses to already established connections. This alteration is dependent on support in the underlying library of Apache HTTPComponents(HC for short) which we have also altered to include support for setting traffic class. Since Synapse is dependent on HC we are, as of the midterm report, trying to get our changes accepted into HC[23]. We have included in our references SVN[24] diff files and included instructions on how to apply them to a newly downloaded version of HC. We are planning to also push our changes in Synapse upstream, but we are waiting for the acceptance of our HC changes first.

7 Implementation

[The specifics of the implementation of the solution.]

8 Testing

[The testing setup and suite. The testing method and how we did the testing.]

8.1 Unit testing

We decided quite early on that we wanted to do unit testing of every piece of code that we would produce, i.e. test driven development. The reason behind this choice is that we think it will result in better code quality. An added bonus is the simplification of integration testing, due to easier discovery of whether a new code addition will integrate with the old code. Also writing the tests first lets us concentrate more on exactly what the methods should do, instead of the content and how it should do it. One of the problems with test driven development however, is the possible bias that could occur, we could end up only satisfying the test and not the actual requirements. This could be countered to some extent by writing more comprehensive tests. Another positive point in favour of unit testing is the requirement we have, which states that the product has to be written in Java where such test are easy to integrate and write using JUnit.

[23] Link to our ticket in HTTPComponents HTTPCore issue tracker [JIRA: HTTPCore Issue 295]

[24] SVN is a centralized content management system used by many of the Apache projects. For more see glossary

8.2 Integration testing

For integration testing, we decided that we wanted to do automated system testing every other week in collaboration with code reviews. The procedure we are going to follow will be coding new features in a separate branch. Once every other week the finished branches will have all their unit tests run thoroughly, followed by a code review of at least one person. Then if the automated system tests are fully operational, they will also be run to look for additional errors which the unit tests can not pick up. This point is likely to change in the future as a two week time interval might be to long given the short implementation period. The advantage of doing this integration testing is better overall code quality, since we will test code before it is used by other parts of the system. Since we are also doing code reviews, people will also gain experience with other parts of the system which they previously had not worked on. This will benefit everyone since knowledge about the system is shared, and it will help in the eventuality of someone getting sick. The advantage of developing in separate branches is the reduced risk of polluting code other people are working on, and a better separation of stable and unstable code.

8.3 System testing

When it comes to system testing, the customer was quite insistent that we test the product thoroughly in an emulated network situation. Since we have had some experience with NS3 we decided that we wanted to do the system testing on it. The advantage with this, is that the customer has already set up some testing scenarios and helper-scripts designed for NS3, which they offered for our use. This will greatly reduce the time needed for setting up the test suite, and it will also give us the ability to have automated tests, which we don't have to monitor or interact with. Another added advantage is easy testing, as we only have to start a script in order to run the whole suite, but that comes at the cost of actually setting up the whole thing. As of the midterm report, we have set quite a lot of time aside in order for us to implement the proper NS3 support we need. To monitor what is happening during the test-runs, our applications will output all important information regarding what is going on, in addition to this we will have a packet sniffer on each end which will capture network traffic. Using this information we should be able to tell a whole lot about what is going on in the network and we should be able to decide whether we have met the requirements or not.

Below are some of the detailed test cases which we will automate on top of NS3. The testing itself will be automated, but in order to get some result from the tests, some human interaction is needed to interpret the output data.

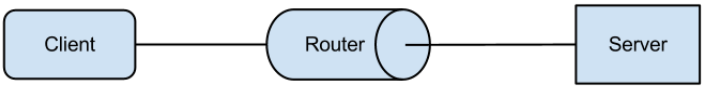


Figure 23: Simple message sending

Simple message sending:

In this test, we want to test the ability of the client and the ESB to communicate. We want to see that the client is able to send messages(message) to the server and get a response back. For monitoring purposes, this test will rely on both applications to log their behavior. In order for us to give this test the green light we must see a message going out from the client then passing through the ESB to GlassFish. Then finally a reply should be sent back from GlassFish to the ESB and then to the client.

Setting Quality of Service:
In this test, we want to test the client and the ESB's ability to set the DiffServ field in the IP header. The first requirement is that the test "Simple message sending" has been passed. For this test to be considered a success, the client has to send a message to the ESB, which is responds with the DiffServ value in the SOAP header. The ESB must at this point have set the DiffServ value in the IP header. The client should then use a service on GlassFish, but this time the IP header must contain the correct DiffServ value. In order to monitor this test, only a packet sniffer located on the client and ESB side needs to be used. The packets must be examined, and the correct DiffServ value must be present in the IP header of all packets, except the first one going out from the client.

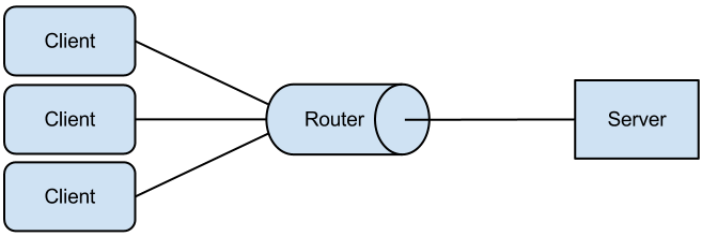


Figure 24: 3 client message sending

Prioritizing messages:
In this test we want to test the ESB's ability to prioritize messages. The scenario will be set up with two clients sending lots of messages in an attempt to flood the capacity of the network. All of these messages should have the same priority, but intermittently, a third client with a higher priority will attempt to send some messages. What we are looking for is that these higher priority messages should be sent out from the ESB before the ones with lower priority and, if necessary, it has to stop some already sending messages. For this test to be successful, we must see some lower priority messages being preempted or held back. To do this, the log file of the ESB must be studied, and there should be some clear indications of one of the requirements.

Changing DiffServ value:
In this test we want to check the ESB's ability to change the DiffServ value after a reconfiguration. The test and the result can be performed and examined the same way as "Setting Quality of Service", but this time the test has to be run twice. One where the configuration has one DiffServ value, and a second run where the DiffServ value has a different value. For the test to be successful, one would have to examine the resulting glspcap files on the server side, and check

each run to see if the two tries have different DiffServ values.

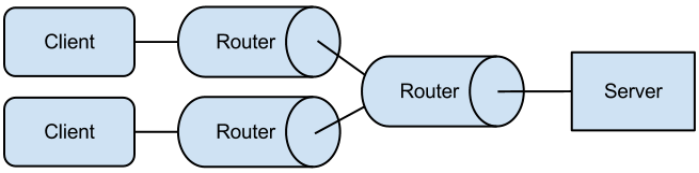


Figure 25: 2 Clients different paths message sending

Multipath server routing:
In this test we want to look at the ESB's capabilities to talk to the MS(Monitoring Service) and understand the routing result. From the MS the ESB should get some routing information about the topology of the network. As you can see in 24, if the link between the server and the first router is not the limiting factor, the two clients should not get in each others way. Therefore, since we get the information about the last router from the MS, the ESB should understand that there is likely no problem and should not preempt any messages. To check if this is actually the case, the ESB will need some time to adjust as it does not get the full picture of the network topology, but after this time, no messages should be dropped from the ESB's side.

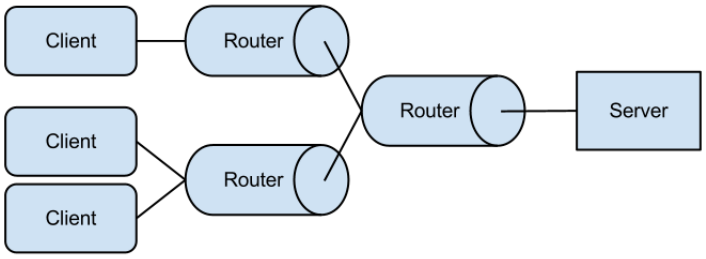


Figure 26: 3 clients where two are competing

Competing clients in a multipath environment:
This test is a compilation of the tests "Multipath server routing" and "Prioritizing messages". For this test we want to make sure that the ESB is smart enough to only preempt the messages going to one of the competing clients. As you can see in the figure 26 there is one client which should not affect either of the two others if the link between the server and the first router is not a bottleneck. This should allow this client to receive messages even though the two other clients are competing for scarce resources. To check that this test is successful, a combination of the clients log files and the server log files will have to be used. If most(over 96%) of the messages arrive at the higher priority client and the third client is not affected then this test is successful.

Competing clients in a low bandwidth scenario:
In this test we want to test that the ESB can manage to prioritize messages in

a network with a joint bottleneck, but with different endpoints. In figure 26, if the link between the server and the first router is the bottleneck, the ESB should after a small initialisation understand that it has to preempt messages going out to all clients in order to let a higher priority client get the service it is supposed to get. The scenario will be set up in such a way that one of the two competing clients will have higher priority than the two others, the two lower priority clients should then send a steady stream of messages which should fill the bottleneck link. The third client should then start sending some messages which must now fill the entire bottleneck link and create a situation where the ESB has to hold back or preempt messages going to either of the two other clients. As before, a combination of the ESB and the clients log files have to be examined.

9 Results

[A thurough presentation of the results of the project. The test results and other interesting findings for this project.]

10 Conclusion

[The summarised findings of the project and presentation of the key findings.]

10.1 Project accomplishments

[Did we reach the goal?]

10.2 Future Work

[What we would have to do in the future to complete or continue this work.]

11 Project Evaluation

[This is the section where we evaluate the project and the process that we have had throughout.]

A Work Breakdown Structure

WBS to be completly implemented later, it is also attached under (B)

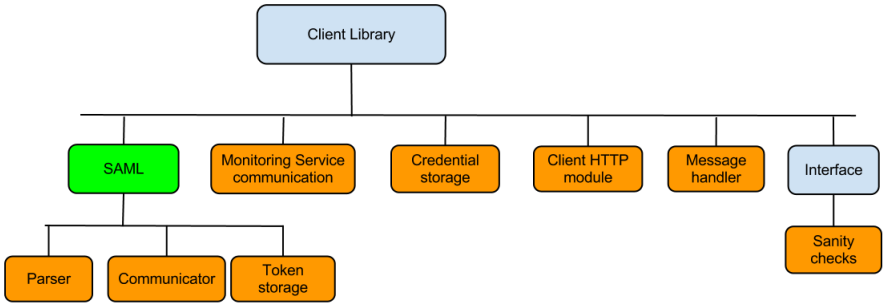


Figure 27: WBS-Client
The work break down structure for the client.

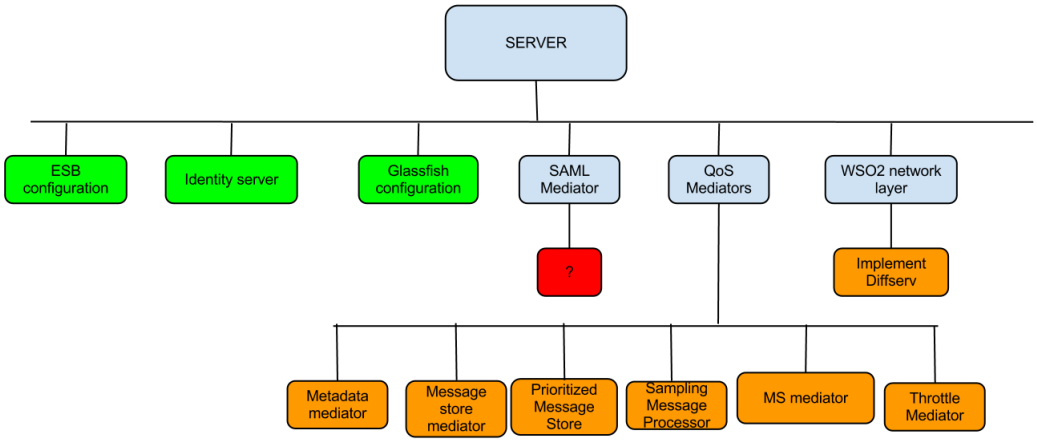


Figure 28: WBS-Server
The work break down structure for the server.

B File Attachments

The files can be unpacked using the pdftk program in Linux with the command: pdftk filetest.pdf unpack_files

Risk List

Gantt Diagram

References