HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Juha-Pekka Laine

Thin-client architecture for Simantics simulation environment

Master's Thesis

Espoo, October 21, 2009

Supervisor: Professor Heikki Saikkonen
Instructor: Antti Villberg, M.Sc. (Tech.)

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author: Juha-Pekka Laine

Title of thesis:
Thin-client architecture for Simantics simulation environment

Date:	October 21, 2009	**Pages:** 5 + 71
Professorship:	Software Technology	**Code:** T-106
Supervisor:	Prof. Heikki Saikkonen	
Instructor:	Antti Villberg, M.Sc. (Tech.)	

Modeling and simulation is currently used in different domains for purposes such as training, education, analysis, and decision support. Currently majority of the modeling and simulation tools are platform specific desktop applications and the models and simulations built with them can only be accessed from a computer with the same tool installed on it.

By making it easier to publish simulation models for a wider audience, brand new applications for simulation could be invented. This approach is now used and a web based simulation environment is developed for publishing and accessing simulation models directly from the web based user interface. This simulation environment contains also the latest communication tools for team work which enables effective development of simulation models in teams.

In this work an architecture for using open source simulation environment Simantics directly from a web site as a thin-client implementation is designed. We will also discuss the scalability of the architecture and all the required components for the system are implemented. In addition, we will measure the performance of the implemented components and the designed architecture is evaluated in a qualitative way to verify its quality and suitability for the system.

Keywords: distributed computing, thin-client, simulation
Language: English

TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Juha-Pekka Laine	
Työn nimi:	Thin-client arkkitehtuuri Simantics simulointiympäristöön	
Päiväys:	21. lokakuuta 2009	**Sivumäärä:** 5 + 71
Pääaine:	Ohjelmistotekniikka	**Koodi:** T-106
Työn valvoja:	Prof. Heikki Saikkonen	
Työn ohjaaja:	DI Antti Villberg	

Mallintamista ja simulointia käytetään useilla aloilla esimerkiksi harjoitteluun, koulutukseen, analysointiin ja päätöksenteon apuna. Mallinnusta ja simulointia tukevat työkalut ovat perinteisesti olleet alustariippuvaisia työpöytäsovelluksia ja mallien käyttö on aina sidottu siihen fyysiseen laitteeseen johon kyseinen sovellus on asennettuna.

Mikäli malleja ja niillä tehtyjä simulointeja voisi helpommin julkaista suuremman yleisön käyttöön voisi syntyä aivan uusia käyttökohteita simuloinnille. Tätä ongelmaa on lähdetty ratkomaan luomalla www-pohjainen simulointiympäristö jossa malleja voi julkaista ja simulointeja voi ajaa suoraan www-käyttöliittymästä. Kyseinen simulointiympäristö sisältää myös viimeisimmät kommunikaatiotyökalut mikä mahdollistaa simulointimallien tehokkaamman kehittämisen ryhmässä.

Tässä työssä suunnitellaan arkkitehtuuri joka mahdollistaa vapaan lähdekoodin Simantics simulointiympäristön käyttämisen suoraan www-sivustolta thin-client toteutuksella. Työssä pohditaan myös arkkitehtuurin skaalautuvuutta ja toteutetaan tarvittavat komponentit toimivan järjestelmän aikaansaamiseksi. Lisäksi toteutettujen komponenttien suorituskyky mitataan ja arkkitehtuuria arvioidaan laadullisella menetelmällä laadun ja tarkoitukseen sopivuuden todentamiseksi.

Avainsanat:	hajautus, kevyt pääte, simulointi
Kieli:	Englanti

Abbreviations and Acronyms

2D	Two Dimensional
3D	Three Dimensional
3G	Third Generation (family of standards for wireless communications)
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASP.NET	Active Server Pages .NET
ATAM	Architecture Tradeoff Analysis Method
AWT	Abstract Window Toolkit
CAD	Computer Aided Design
CPU	Central Processing Unit
DEVS	Discrete Event System Specification
DHCP	Dynamic Host Configuration Protocol
DOM	Document Object Model
EDGE	Enhanced Data rates for Global Evolution
FPS	Frames Per Second
GIF	Graphic Interchange Format
GPRS	General Packet Radio Service
GPU	Graphics Processing Unit
GUI	Graphical User Interface
GZIP	GNU zip
HTML	Hypertext Markup Language
OS	Operating System

RMI	Remote Method Invocation
SaaS	Software as a Service
SMILE	Structural Modeling, Inference, and Learning Engine
SLA	Service Level Agreement
SVG	Scalable Vector Graphics
TCPTE	Thin-Client Applications For Limited Devices
VNC	Virtual Network Computing
VRML	Virtual Reality Modeling Language
WLC	Weighted Least Connections
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language

Contents

Chapter 1

Introduction

Modeling and simulation is currently used in several domains such as manufacturing, military, and health care. It can be used for purposes such as training, education, analysis, and decision support. Simulation also plays an important role in business process management, and a growing number of business management software vendors offer simulation support for their products. (April et al., 2006; Moradi et al., 2008)

Currently simulations are built with tools that are tailored for that specific use only, and each domain has its own ways and tools to create simulations. By creating a common, easy to access environment for developing, publishing, and accessing simulations, we can make it easier to access and use simulations in various applications. This kind of simulation environment can create brand new applications for simulation especially among the general public, and it could create new possibilities for applying simulation in commercial use also. To make the system as easy to access as possible, the system should be implemented using common web technologies and it should be freely accessible from the Internet.

Providing applications through Internet as a service has become increasingly popular during the past few years. Software as a Service (SaaS) refer to this kind of distribution method, and it has many advantages compared to traditional way for distributing software. In SaaS model, the software can be priced on-demand, and the software is hosted and maintained by a service provider. In addition, according to Choudhary (2007) the quality of a SaaS product is likely to be better than the quality of a traditional product. Because of these advantages, SaaS model is usually cheaper and easier for the customer than the traditional way of licensing software. (Turner et al., 2003)

When software is provided as a service, it is usually delivered through the Internet which sets constraints for the software. Most constraints are caused by limited network bandwidth, but security is also an issue. Because of limited network bandwidth, the size of the software should be minimized and the application should be made as

lightweight as possible, since the users usually do not want to experience any delays. The most common method for delivering software as a service is to implement the software as a web application by using only common web technologies, thus making the software accessible with a web browser. This approach is also used in this thesis. (Choudhary, 2007)

This thesis is part of a project called Simupedia, in which a web-based simulation environment is created. Simupedia provides tools for creating, publishing, accessing simulations, but it also provides the latest communication tools, hence making the site a virtual workroom. This kind of application is not a new idea, Henriksen et al. (2002) presented a concept of a web based simulation center which has many similarities to our service. However, we will focus more on providing simulations for general public and for non-professional commercial users who are not that familiar with simulations, but do use ready made simulation models in their businesses. For this purpose, a web-based simulation environment is ideal, and our task is to bring simulations to the web where they can be accessed as easily as possible.

The goal of this thesis is to create an architecture for a simulation player that can be accessed from a web page. The player can be used to quickly access a simulation model and to change parameters of the model to create an own experiment. However, the tool is not for creating or editing models, but these functionalities are provided by a tool called Simantics, which is an open source software platform for modeling and simulation. The simulation player will be heavily based on the Simantics, meaning that it is a thin-client user interface for Simantics, and can be used to access ready made simulation models and to change parameters on the model to create a custom experiment. (simantics.org, 2009)

This work focuses on creating a scalable architecture for the thin-client implementation of Simantics, including the optimal distribution between the server and the thin-client, and the server architecture which scales up to serve a number of concurrent clients. However, this thesis will not pay attention on the Simantics internal architecture that uses a custom database server, even though this architecture might not be scalable enough. In addition to the design of the architecture, we will implement the designed components and measure if we can achieve the performance requirements for the system. Finally, the architecture will be evaluated against the requirements by using Architecture Tradeoff Analysis Method (ATAM).

1.1 Structure of the thesis

This thesis is divided into seven chapters as follows:

Chapter 1 is the introduction providing background and motivation for the work.

Chapter 2 reviews the main technologies related to the thesis topic.

Chapter 3 elicitates and analyses the requirements for the architecture.

Chapter 4 focuses on creating the architecture for a thin-client implementation of Simantics.

Chapter 5 analyses the need for distributing the server side computing and describes the server side architecture.

Chapter 6 evaluates the designed architecture against the requirements by measuring implemented features and analyzing the architectural plans of those components that haven't been implemented yet.

Chapter 7 concludes the study and gives some ideas for further investigation

Chapter 2

Technology Review

This chapter presents the basic concepts behind the thesis topic and the latest studies related to them. This information is crucial for designing the architecture for the thin-client system, but is also useful background information while gathering the requirements for the system.

2.1 Modeling and simulation

In order to develop a user interface for modeling and simulation tool, it is important to know how simulation models are visualized and what are their characteristics. However, it must be remembered that we do not develop a full featured model editor, but a model and simulation viewer. We will focus on visualizing simulation models and simulations made with the model, and leave most of the tools and interactive features found in the model editor unimplemented.

Simulation refers to solving a mathematical model of a physical process. The mathematical model is called simulation model and it is a static configuration consisting of components and links between them. The graphical presentation of a simulation model is like in any model in computer aided design (CAD). The model can be either two dimensional (2D) or three dimensional (3D) and is usually presented in vector graphics. In addition, some softwares might have support for additional vector and raster graphics components, such as background images. These components are not usually part of the actual simulation model, but might be used to present the results of a simulation or for example to help the user understand the model.

Simulation can be either steady-state or dynamic. In steady-state simulation, all the properties of a system are constant with time. In dynamic simulation, a time-dependent data series is produced instead of a single set of result values. Steady-state simulation

produces a set of result values that can be visualized as such or with the simulation model by using static values and graphical components. In general, the results of a steady state simulation can be visualized with a single vector graphics image that consists of static vector components.

Visualizing dynamic simulation is, however, a more difficult task. The set of result values varies by time, meaning that the numbers and graphics used to visualize the results of the simulation changes constantly. An example of this kind of simulation is a model of two water tanks and a pump between them. When the pump runs, the water level gets lower in one tank and higher in another. In addition, we could display the how much water is in each tank by using numbers and visualize the rotation of the pump. Dynamic simulation requires dynamic graphics that will cause constant interaction between the client and the server, hence we have to optimize the solution for dynamic graphics.

In most cases the simulation model and the simulation results can be presented in vector graphics which consumes less network bandwidth than raster graphics, and thus it is enough that the client is capable of visualizing vector graphics. As a simplest example, a working solution could be to implement a client that simply displays vector graphics images that comes from the server, however, this solution would be far from optimal. Even though support for raster graphics is not required, it should be implemented if the architecture makes it possible.

(Lehtonen, 2007)

2.2 Thin-client

The term thin-client refers to a software layer that supports graphical user interface on a local machine while the actual application program is executed on a remote machine. The benefits of this architecture are low management costs and low hardware requirements on a client side, thus making it possible to run heavy applications also in low performance devices. The application itself runs on a high performance server which is usually hosted by a service provider who will also perform the management tasks, such as software updates. The main drawback of the thin-client architecture is the need for a fast network connection, especially when using highly interactive graphical applications, such as CAD and image processing. In these kind of applications, the delays experienced by the user are increased by the need to transfer large images and vector graphics through network connection. (Coulouris et al., 2005)

While desktop PC:s and laptops are already equipped with high enough computing power for running almost any type of applications, we do have increasing amount of low performance devices such as mobile phones and netbooks that are capable for run-

ning graphical applications but have limited amount of computing power. When the desktop application is designed for the latest PC hardware equipped with high computing power, the need for a thin-client implementation becomes even more important. The main benefit of the thin-client architecture presented in this thesis is that it allows users to access the simulations without downloading and installing the whole simulation software, thus making the simulation models accessible from almost anywhere.

By locating the actual simulators to the server-side we are enabled to provide also content based on non-free simulation solvers. Because the solver is running on the server-side it is never distributed to the client and thus the client does not have to purchase a license for the solver. Even though this arrangement does not allow us to use any solver on the server side, we have better possibilities to get licenses for the solvers when we do not redistribute the simulation solvers to our customers.

It also has to be noted that even if Simantics is platform independent Java software, some of the solvers integrated into Simantics are platform dependent, thus will not work on every computer as such. For those who actively model new simulation models this might not be a problem since they probably have suitable platform for that work, but for those who occasionally need access to ready made simulation models and results this might be a major problem. In any case, to integrate the heavy simulators seamlessly to the website, we need a lightweight client application, and thus the thin client is the only feasible solution.

A number of thin-client architectures have been designed and implemented during the years. The following sections presents the most common ways to implement a web based thin-client application.

2.2.1 Ajax

Asynchronous JavaScript and XML (AJAX) is a method for building interactive web applications. It is not a technology or a programming language, but a set of technologies that work together to create a new development model for web applications. While traditional HTML or XHTML web sites provides interaction only in a form of loading a new page, AJAX brings a desktop application like interaction to web site by integrating XHTML, JavaScript, DOM, and different data transfer technologies to a traditional web page. With AJAX, it is also possible to display dynamic vector graphics on a web page by using the native vector graphics support that the latest browsers have. One of the main benefits of AJAX is that is does not require any additional plug-ins to browser, hence the application is accessible from any modern computer that is equipped with a basic web browser. (Zepeda and Chapa, 2007; Eick et al., 2007)

Even without using the latest AJAX technologies, it is possible to implement a web

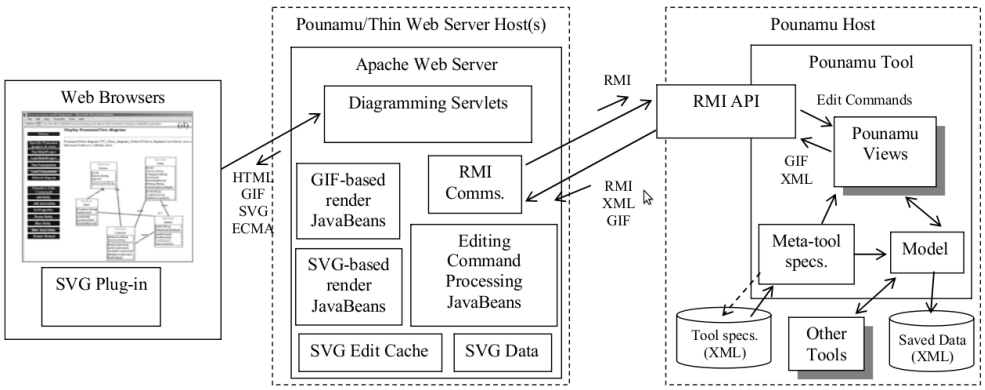


Figure 2.1: Architecture of Pounamu thin-client (Cao et al., 2005).

based diagramming tool by using XHTML and basic JavaScript. Cao et al. (2005) describes a generic web-based user interfaces for diagramming tools. In the article the authors designed a web-based system called Pounamu for viewing and editing diagrams that consist of GIF images, Scalable Vector Graphics (SVG), and 3D VRML models. Pounamu is a thin-client implemented with XHTML and Javascript. The server side is implemented using Java Servlet technology and Remote Method Invocation (RMI). RMI is used in the Java Servlet implementation to communicate with the original application. The architecture of the Pounamu is presented in figure 2.1.

The main benefit of this approach is that is uses only basic web technologies, and the application can be used with a very basic web browser. Because all the advanced mobile phones are equipped with such a browser, the application can be used with a mobile phone, and thus is accessible from anywhere at anytime. Although the user interaction in Pounamu is not as good as in in a desktop application, it proves that it is possible to create a thin-client by using web technologies only. However, because the solution is a few years old, it does not use all features and advantages of the latest AJAX web technologies. The latest AJAX toolkits have support for vector graphics, which enables possibility of creating better and more interactive user interface than Cao et all. have created. An example of a such application is GeoBoost developed by Eick et al. (2007). Although the analogy to Simantics is not as clear as in Pounamu, there are many similarities to our project, such as dynamic vector graphics based web user interface.

Johnson and Jankun-Kelly (2008) have compared performance of different technologies for displaying vector graphics in a browser. In the comparison the performance of a AJAX based solution is significantly weaker than the performance of a Java based solution. However, web technologies and browsers evolve quickly and AJAX based solution for vector graphics is likely to become feasible in the future, but currently a

Java based solution seems to provide better performance.

Although AJAX is a promising approach and is suitable to be used in an application like Simantics, it would require major changes to Simantics, since it does not use vector based format internally for displaying graphics. Even though most of the component symbols in Simantics are in vector format, all the dynamic graphics is painted directly with Java 2D API. It has also been pointed out that using vector graphics as an internal graphics format in Simantics will cause significant performance loss. By creating separate vector graphics painters for the thin-client could be a working solution, but maintaining separate painters throughout the development of Simantics would require too much effort. However, if possible, the architecture should provide an interface that could be used in the future to fetch the vector graphics presentation of a simulation model from an AJAX application.

2.2.2 Java

Implementing the thin-client in Java is noteworthy idea since the Simantics application is also implemented in Java, thus the integration should be quite easy. Java application can also be integrated seamlessly into a web site by using Java Applet technology. Canfora et al. (2006) has used this approach to develop a thin-client solution to existing applications for mobile devices. Their solution, called Thin-Client Applications For Limited Devices (TCPTE), uses Java 2D Abstract Window Toolkit (AWT) for remote visualization. In Java 2D all painting is performed by using Graphics2D class methods. The idea is to serialize all calls to Graphics2D methods and send them to the client instead of displaying locally. The client can then unserialize the method calls and call them on local graphics device. This approach is completely transparent for the application developer, and can be used to easily develop thin-client application from an existing desktop application. This approach is interesting because it is made for devices with low computing power, and still it is compatible with Java AWT. Because Simantics uses Java AWT for model visualization, this approach would require only small changes to current implementation of Simantics.

Although TCPTE client is designed for mobile devices, this solution works as well in Java Applets that can be integrated into a web page. Java applets uses also Java AWT for displaying graphics, hence the TCPTE solution works in applet without any changes. On the contrary to AJAX, Java Applet requires browser plug-in in order to work. These plug-ins are available for the most popular browsers and operating systems, but it still requires extra actions from the user to get it installed. However, the performance of a Java based solution for displaying vector graphics is significantly better than the performance of a similar AJAX based solution, and thus the user is able to get better user experience by installing the plug-in (Johnson and Jankun-Kelly,

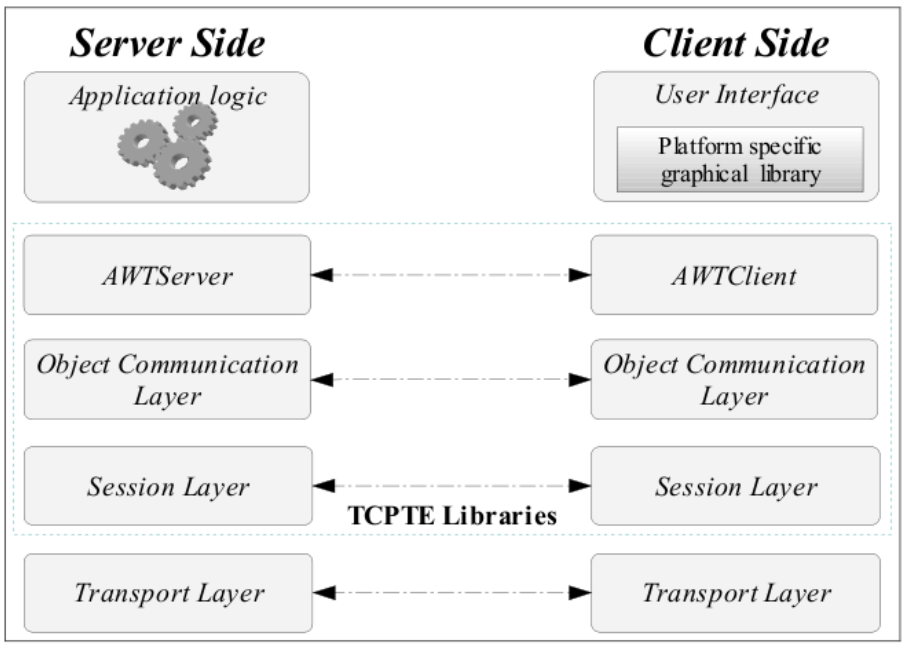


Figure 2.2: TCPTE Architecture (Canfora et al., 2006).

2008). The architecture of the TCPTE is shown in figure 2.2. As the figure shows, TCPTE is actually a wrapper class for serializing the methods in Java AWT. (Sun Microsystems, 2009)

The problem with the Graphics2D serialization is that it consumes relatively much bandwidth, because every single paint command must be serialized with parameters. The solution can be optimized by compressing the data and using some sort of cache, but the solution would not still be optimal. However, the implementation of Graphics2D serialization is quite straightforward, and thus should be implemented and evaluated.

2.2.3 Flash

Adobe Flash is a technology for creating standalone multimedia presentations and applications to web sites. It is especially suitable technology for creating vector based graphical applications for web. Although Flash is designed for standalone applications, it can communicate efficiently with server side applications by using Flash Remoting technology (Macromedia, 2002). With Flash Remoting technology, Flash is suitable for creating light weight user interfaces for applications that are executed on a remote machine. Like Java Applet, Adobe Flash requires browser plug-in in order to work,

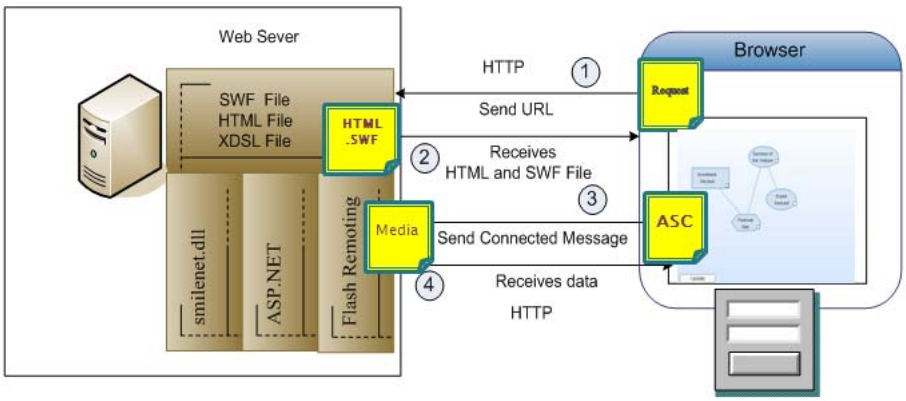


Figure 2.3: Architecture of SMILE Flash visualization system (Ketterl et al., 2007).

but because Adobe Flash is used in many popular web sites, the plug-in is already installed in many computers. Compared to Java Applets, Flash has better support for vector graphics and it is more suitable for creating graphical user interfaces than Java. However, network communication technologies in Flash are not as advanced as in Java. (Ketterl et al., 2007; Tungkasthan et al., 2008)

Tungkasthan et al. (2008) describes an example how Flash client applications can be integrated into an existing application that is running on a server side. They develop a web user interface for a Structural Modeling, Interface, and Learning Engine (SMILE) by using flash technologies. The analogy to Simantics is clear, both applications have vector graphics diagram as model configuration and uses server to calculate data based on the diagram configuration. On the server side, Tungkasthan et.al. have Flash Remoting and ASP.NET. Flash Remoting is used to communicate with the client and ASP.NET is used to execute the required tasks in the SMILE core. This approach is described in figure 2.3.

In this kind of Flash based solution, the client is responsible for rendering the graphics, thus moving part of the computation away from server to the client. However, in order to work, this approach would require major changes to the current Simantics implementation. Because the benefits compared to Java applets are minor, there is no reason to implement the thin-client in Flash.

2.2.4 Virtual Network Computing

Virtual Network Computing (VNC) is a more generic approach for thin-client computing than the ones we have presented above. In VNC, graphics is actually transferred to the client in a video stream, hence it can be used to display any kind of graphics.

For this reason, VNC is widely used for remotely accessing operating systems, and applications running on it. However, to achieve good usability, VNC solution requires broadband connection, and thus is not very feasible solution for systems that should be accessed using mobile devices. (Vankeirsbilck et al., 2008)

What makes VNC an interesting solution, is that a service called NanoHUB uses it to integrate dynamic simulators into a web site. NanoHUB has many similarities to Simupedia. It is a nanotechnology web community that provide resources and tools related to nanotechnology. For example, they provide custom simulators and computing power for those. Some of the simulators require heavy graphical calculation, thus the graphics must be rendered on the server side. Contradictory to Simupedia, the NanoHUB simulators are custom applications that include custom user interfaces and application logic, hence the application must be executed as separate processes on the server side. For this kind of system VNC is ideal, since it can be used to access any graphical application remotely. (Qiao et al., 2006)

VNC would not require any changes to Simantics, but its bandwidth requirement is relatively high, and in order to work smoothly, at least broadband connection is required. In addition, VNC server side implementation is heavy because the server must perform the actual rendering and then compress the rendered data before it can be streamed to the client. This is a major drawback in a VNC solution, since for example in a Java based implementation described in section 2.2.2, the server does not perform rendering at all. (Yang et al., 2002)

2.3 Load balancing

Running a large simulation model requires significant computing resources and the small models are not lightweight either, hence running multiple simulation models for a large amount of concurrent users requires large amount of computing power, and a single physical server cannot not have enough computing power for serving all the users. In this kind of situation the load must be distributed among multiple server machines, but to efficiently distribute the load between the servers, a load balancer component with a tailored load balancing algorithm is required.

Load balancing is about exploiting available computational resources by distributing load between multiple sites. In our case, we want to exploit multiple server machines to be able to provide good enough performance for the clients. The way how the load is distributed across the servers depends on the algorithm used.

A number of different algorithms have been designed to distribute the load evenly among a large set of servers. There is no single algorithm that is good enough for every purpose, but the algorithms usually require tailoring before they can be utilized

to balance load in a specific application (Kunz, 1991). If we compare our system for example to web site load balancing, we have much longer sessions and each connection requires much more computing power. When in web, requests are simple page requests that will take less than a second to perform, and the amount of concurrent connections to servers can vary significantly each second. For example the following load balancing algorithms have been widely used in systems similar to what we are developing:

- Round robin
- Weighted round robin
- Least connections
- Weighted least connections

Round robin distributes incoming connections to each server in circular order, meaning that the algorithm selects always the next server in the circular list for the incoming connection. This algorithms works well if the sessions are short and cause equal load to the server. However, with varying length sessions, round robin can cause load to be distributed unevenly. On a heterogeneous system, basic round robin algorithm does not take into account varying computing capabilities of the servers. For servers with different capabilities, weighted round robin is a better choice, because each server can have a weight that indicates its computing capabilities, and more powerful servers can get more connections than the less powerful ones.

If we assume that each session consumes CPU and memory resources equally, least connections algorithms is better choice than round robin. Least connections algorithm requires that we always know how many connections there are to each backend, thus the implementation is more complicated than the implementation of a round robin algorithm. When a new connection is accepted, the algorithm searches a server that has least connections and redirects the incoming connection to that server. For this reason the connection initialization takes longer than in round robin, but because we have long sessions in our application, the small increase in the connection initialization time is not visible for the user. Least connections algorithm works well in systems where sessions are long and the duration of sessions varies.

Because it is probable that we will end up in a situation where we have heterogeneous server environment, we do not want the connections to be distributed equally, but to take the capabilities of different servers into account. For this purpose, an enhanced least connections algorithm called weighted least connections (WLC) has been designed. Like weighted round-robin, WLC has weight for each server, and the weight will be used in calculations when a new connection comes in. For example, a server with weight two will get twice as much connections as a server with weight one.

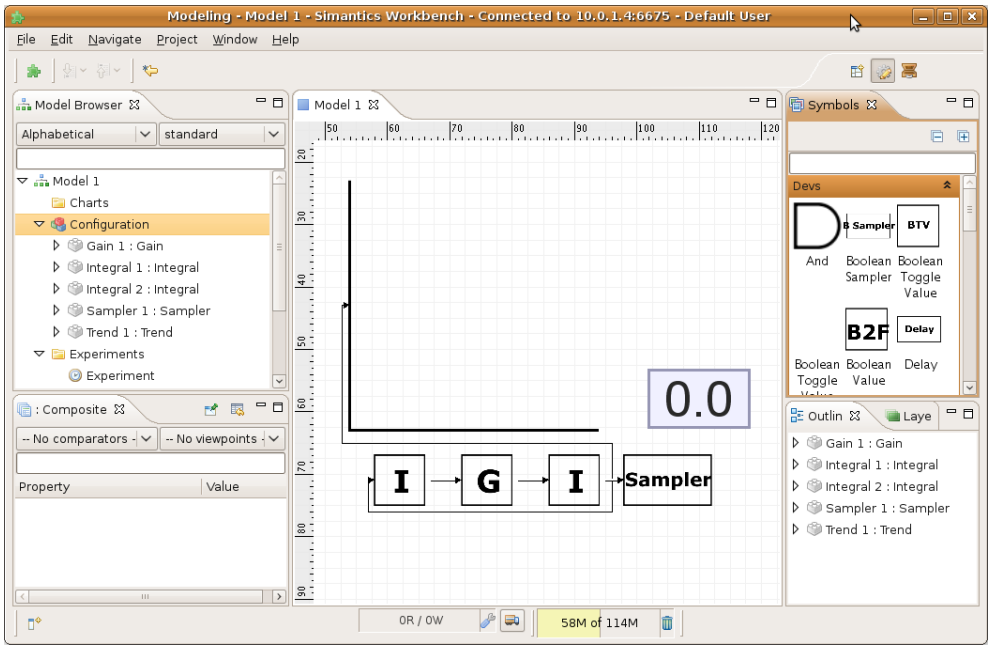


Figure 2.4: Simantics simulation environment.

Because it is difficult to measure how much server resources are currently used and even more difficult to predict how much resource consumption will vary, we will assume that each client session requires equal amount of CPU and memory. By using this assumption, we do not have to measure and predict CPU and memory load, but we will make sure that client connections will be distributed evenly with the servers. We know that some sessions will require more resources than others, but we can be quite confident that these resource consuming sessions will be divided roughly equally among the servers, and thus the number of concurrent connections will reflect the server load quite well. A more detailed example of WLC will be presented in the chapter 5, where the load balancing algorithm is designed.

(Youn and Chung, 2005; Youn, 2005)

2.4 Simantics

In this thesis we will create a thin-client solution for an application called Simantics which is an open software platform for modeling and simulation that heavily utilizes the concept of semantic data models. It is internally based on a client-server architecture, but to avoid confusions between different server concepts, the Simantics server component will be referred as database. The term database actually describes servers

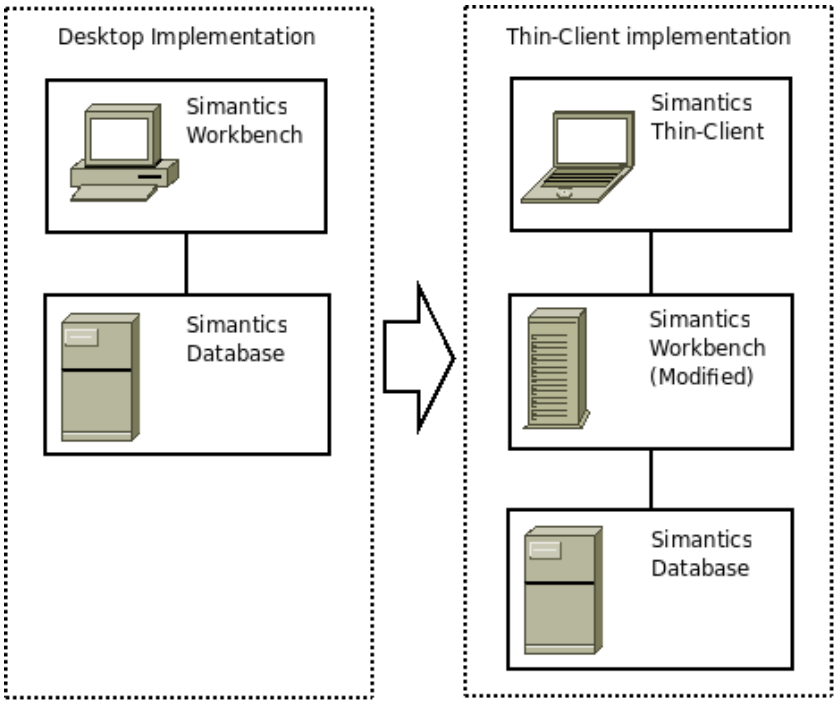


Figure 2.5: Current Simantics components and the components needed in the thin-client implementation.

functionality better, since its task is to store semantic data related to the simulation model.

Simantics simulation environment contains for example tools for viewing and editing simulation models, for running simulations, and for managing simulation projects. Figure 2.4 presents the Simantics workbench model editor.

The client side of the Simantics is based on the Eclipse platform and it contains the application logic that will also be needed in the thin-client implementation. For this reason, we have to modify the new server component from the Simantics client side and keep the database component in the architecture. This modification is presented in figure 2.5. In the final system, we will have three main components: the thin-client, the server and the database. (simantics.org, 2009)

Because the thin-client does not provide tools for editing simulation models, we must provide a full featured Simantics workbench desktop application in Simupedia web site for those who are willing to edit the models. Simantics workbench requires access to the same database where the thin-client servers are connected to. This makes the architecture for the thin-client system more complicated. A preliminary architecture containing all the main components is presented in figure 2.6. However, the database

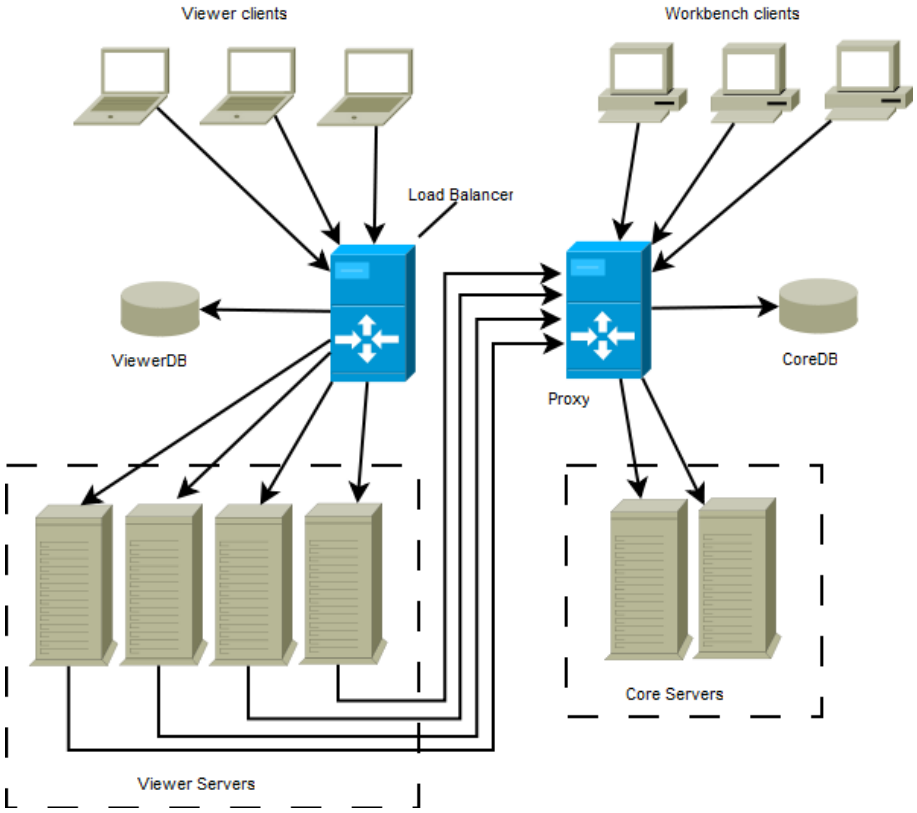


Figure 2.6: Preliminary architecture for the thin-client system using Simantics.

servers and the database proxy presented in the picture, are not a part of this work. In this work, we can assume that viewers can connect to a database by using a single static address, even though a single database instance would not be capable for serving the significant number of concurrent connections in reality.

Chapter 3

Requirements specification

While analyzing the requirements for the thin-client architecture, three perspectives must be taken into account. First, the Simantics way to present simulation models determines what kind of data the thin-client must be able to display. Second, many non-functional requirements comes from the users perspective, since a good user experience is required. Third, we get many cost related constraints from business side, hence the architecture must be cost-efficient. In this chapter we define the requirements for the whole thin-client architecture. We will define what kind of data the thin-client should be able to display, how the user should be able to interact with the application, and the minimum system requirements for running the thin-client. In server side, we will define for example how scalable the system should be and how much one user can cost for the company.

3.1 Constraints

- Windows XP, Windows Vista, Mac OS X v10.5 and the most common Linux distributions must be supported.
- Must work and be usable on a basic netbook (Intel Atom 1.6GHz, 512MB, 1024x600)
- Must be usable through 384kbit/s 3G connection
- Must work somehow through 384kbit/s EDGE connection
- Only common web technologies can be used, such as Flash, Java or Ajax

Even though the 3G and EDGE bandwidths are equal, the bandwidth of EDGE connection is theoretical, hence the actual bandwidth of EDGE is somewhere between 150kbit/s and 200kbit/s (Negreira et al., 2007). A solution that requires as few changes to Simantics as possible should be preferred. In addition, the solution should be striven to be made such that it can be made accessible from mobile phones with small development effort.

3.2 Requirements for the thin-client architecture

In general, the thin-client must be capable to display vector graphics from the server and to send input events back to the server. Even though it is possible that the simulation model contains raster graphics, all the important components should be vector based, and thus raster graphics can be ignored by the thin-client if the architecture is not capable of displaying it. Some of the components in the diagram are static, hence they do not change during the session, but some components are dynamic such as trends and input/output widgets. However, the user must be able to zoom and pan the diagram, which means that translations (position and zoom) of the static components can change.

As noted earlier, the thin-client must be able to display the graphical presentation of simulation models. The characteristics of a such model has been described in the previous chapter, but the basic components that the thin-client must be able to display are:

- Vector graphics
- Static text and numbers
- Dynamic input and output widgets
- Dynamic trends
- Rulers and background grid
- Raster images (optional)

The Simantics workbench uses Java2D API for drawing, hence basically by supporting the Graphics2D interface for drawing, we are able to display everything that is needed. However, everything that is drawn could also be presented in vector graphics.

Thin-client should support the following functionalities from Simantics:

- Display simulation model diagram

- Navigate between diagrams
- Pan diagram
- Zoom diagram
- Change values of input boxes
- Navigate between diagrams using clickable links
- Play, Stop and Pause simulation
- Switch to fullscreen and back

This means that editing model should not be supported, but browsing the model and changing the parameters of the simulation and controlling the simulation should be possible. The interactive functionalities are the key usability issues in an application, and thus to provide smooth interaction, latency in the interactive functionalities must be minimized. This is especially crucial in pan and zoom functionalities.

In addition to the functional requirements listed above, the architecture should fill the following non-functional requirements:

- The application and the diagram must be loaded in less than 10 seconds *.
- Average bandwidth consumption from the server to the client must be less than 100kbps.
- Average bandwidth consumption from the client to the server must be less than 50kbps.
- Maximum allowed latency experienced by the user is 200ms (from input to visual feedback) *.
- The user must be informed of any application error, and the application must provide information how to proceed.
- Thin-client implementation should not need to be changed even if new plug-ins are added to Simantics.

(* Using the best quality 3G link available)

Most of these non-functional requirements must be evaluated against the constraints listed earlier in this section.

3.3 Requirements for the server-side architecture

The server must provide following functionalities for the client when requested:

- Attach multiple clients to a common experiment.
- Run experiments on background, and attach running experiment to a diagram.
- Provide a list of available diagrams by model name.
- Access control for non-public models.

For administrative purposes, the server must provide following functionalities:

- Maximum amount of concurrent connections must be possible to limit.
- Maximum memory usage per client must be possible to limit.
- For certain user groups, it must be possible to address a dedicated server for running the simulations.
- Connections from specific IP addresses must be able to block (if harmful client is detected).
- The system must write log entries to a centralized log system.
- If a server machine fails, it must be possible to be replaced with a new server on the fly, so that the system can accept connections to any models at any time. However, the clients attached to the server that failed, does not have to be seamlessly switched to a new server, but they can be requested to reconnect.
- Incoming connections must be able to be guided to a specific machine, or away from a specific machine, to make it possible to schedule shutdown for a server without interrupting the users. (Redirect connection to another machine and wait until active users closes the connection before shutdown)

On the server side it is important that we are able to serve as many concurrent users as possible with minimum cost but still we have to provide good user experience. For example these factors are taken into account in the non-functional requirements of the server-side architecture:

- Distribution must be transparent for the end-user.

CPU	AMD Phenom 9550 2.2GHz quad core
RAM	4GB DDR2
GPU	GeForce 8200
Network	1000Mbps ethernet

Table 3.1: Specification of a reference server. We will use commodity hardware on the server side.

- One server machine (described in table 3.1) must be able to serve 100 concurrent users.
- 100 concurrent users must be possible to serve through 10/10Mbps network connection.
- The system must scale up linearly, thus with twice as many servers and network bandwidth, it is possible to serve twice as many concurrent users.
- The system should work even if a single server machine fails. However, the users on that server can be requested to reconnect.
- Deploying new server processes with custom configuration must be possible, i.e., heterogeneous backends.

3.4 Reference models to be used in the measurements

The computing and bandwidth requirements may vary significantly between different simulation models and solvers. For this reason, we need to describe reference models that will be used in calculations. An average model is difficult to describe, instead we will define one simple model and one more complicated model. Currently the only public simulation solver integrated to Simantics is a solver based on Discrete Event System Specification (DEVS). DEVS is a formalism for modeling and analysis of discrete event systems (Zeigler et al., 2000). Even though the models made with DEVS are relatively simple and thus do not exemplify an average simulation model, we are forced to use these models as a reference in our measurements.

The simple model is described in the figure 3.1. It is a very basic model and it simply displays sine values in a trend. The model has three components: integrator, gain, and another integrator. In addition, it contains dynamic trend and sampler components to visualize the sin values. Even though fully static model would be even simpler, it

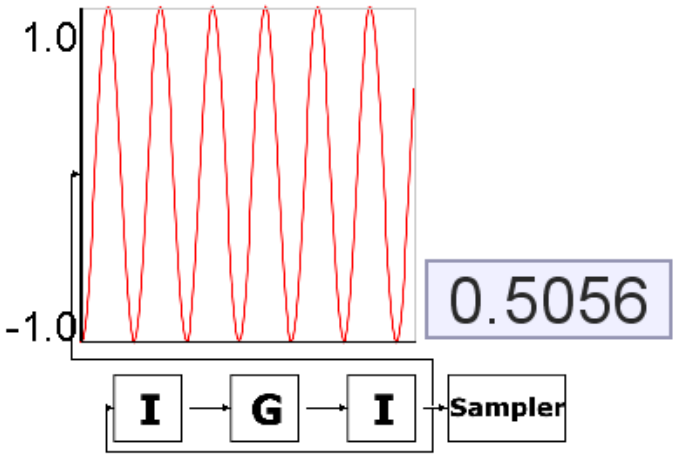


Figure 3.1: Simple model configuration. When the experiment is running, the trend component will display sin curve and the monitor will display the sampled value.

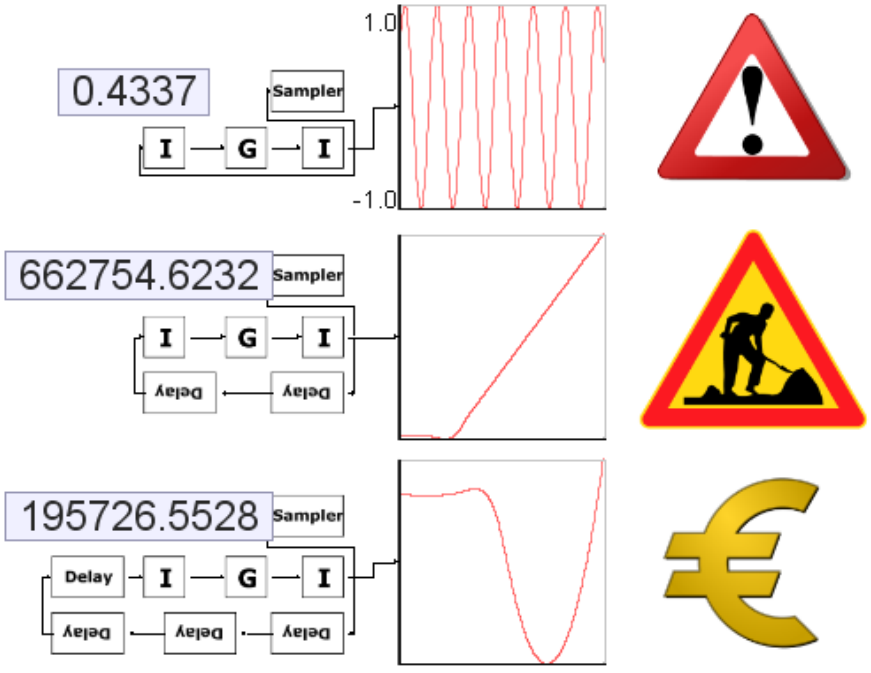


Figure 3.2: Advanced model configuration. When the experiment is running, all trends display curve and monitors display sampled values.

would not cause any traffic between the client and the server during the simulation, and the measurements would not be that interesting.

The figure 3.2 shows the advanced model configuration. The advanced model is actually three different models. It contains the simple model, and two modifications of the simple model. Because DEVS models do not consume that much computing power, the advanced model is designed to consume more bandwidth than the simple model. For this reason, the advanced model has six dynamic components and additional SVG graphics. The SVG images are randomly chosen from the Internet, since SVG graphics are usually used to decorate the model, and usually the used images are searched from the Internet. Because SVG images are static components, they consume bandwidth only when the model is loaded, but the dynamic components cause bandwidth consumption during the simulation.

Chapter 4

Thin-client architecture

In Simupedia, we will use Simantics simulation platform to perform the actual simulations. However, Simantics is quite heavy software and it supports external simulation solvers that may not be compatible with the operating system the user has. The solvers are responsibe for the actual calculations, and thus are also resource intensive. For these reasons, a lighter user interface is required to make it easier for users to access the simulation models. The solution is to build a web browser based thin-client user interface for the Simantics simulation platform and locate the heavy and platform specific software components to external servers, thus making it possible to run the simulations from the web site.

This chapter describes how the user interface is currently implemented in the Simantics simulation environment and how the same user interface can be implemented as a thin-client. In this chapter, we assume that we have only one client and one server. Next chapter will concentrate on scaling up the architecture for multiple concurrent users.

4.1 Technical challenges

Simantics architecture is very flexible and extensible, but for this reason the internal interfaces are vague. One of the design choices is that the architecture should give as few limits as possible for the programmer. On a very high level, Simantics is a Java application based on the Eclipse platform which can be extended with plug-in components as shown in figure 4.1.

The Simantics user interface is based on Eclipse components, but the diagram view is implemented in a custom way. The diagram view is the white area surrounded with rulers and a grid background shown in the figure 2.4. The diagram view displays the model configuration and operating interface. In the thin-client user interface, it

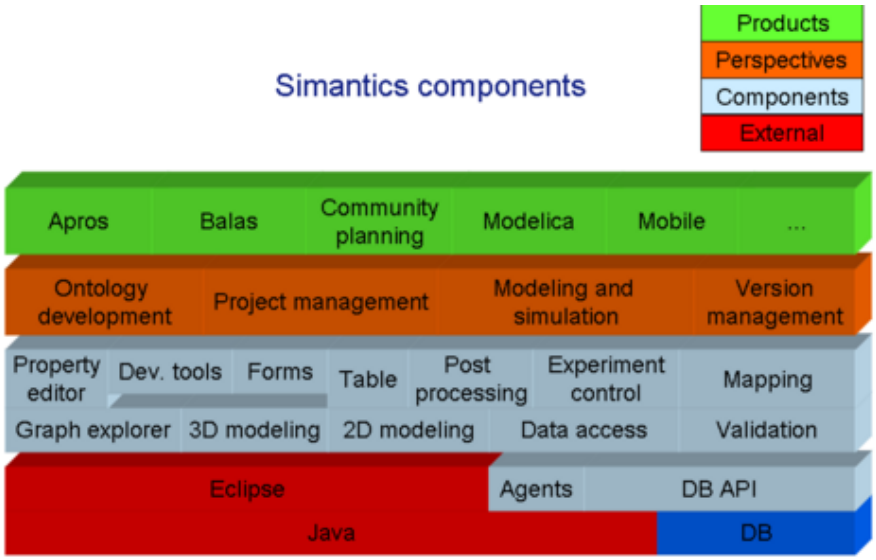


Figure 4.1: Simantics component view. (simantics.org, 2009)

is enough to display only this diagram view and leave out rest of the user interface components. In the diagram view, the user must be able to navigate and change values of the input components. The thin-client does not have to provide support for editing the model configuration or for using all the advanced features of Simantics. A detailed list of the requirements for the thin-client is described in chapter 3.

In a basic thin-client solution, the server sends the graphics to the client and the client sends the input events back to the server. The main constraints for the thin-client architecture are network bandwidth and latency. In addition, as much computing as possible should be performed on the client-side because current computers generally have unused computing resources and we have only limited resources on the server side. However, it must not be forgotten that the idea of a thin-client solution is to provide lighter user interface for the simulation software. Hence, it must be considered how much computing is performed on the client side and how much on the server side.

Many interfaces in Simantics are vague, and so is the diagram painting interface, hence the programmer can access to almost any Simantics resources directly from the paint code. This means that there is no clear interface that could be used to transfer graphics from the server to the client side, except the Java 2D API. Transferring rendered graphics with VNC solution would also be possible, but it is far from an optimal solution, and thus its use should be avoided. The only solution left is to use the Java 2D interface because we do want to avoit major changes to Simantics. As described earlier, the Java 2D Graphics2D interface operations can be serialized through socket stream to the client, and the solution has been proved to work even with limited devices by

Canfora et al. (2006). Because this solution is promising and does not require much effort, we decided to implement and evaluate it with Simantics. The solution turned out the be a working one but the bandwidth requirement was relatively high on models with much interaction. Details about the Graphics2D serialization is described in the next chapter. However, we had the possibility to redesign Simantics painting interface we managed to create a better solution which is described later in this chapter.

As a side note, we do not pay much attention to input event handling, since the events from the client to the server does not require much bandwidth and the implementation is quite straightforward. However, there is a few design issues for minimizing the latency described in the final architecture section. The key problem is to transfer the graphics from the server to the client.

4.2 Java Graphics2D serialization

Canfora et al. (2006) created a thin-client solution based on Graphics2D serialization over network socket to allow access to desktop applications from mobile devices. Graphics2D is a Java class containing 39 own methods 45 methods inherited from Graphics class (Sun Microsystems, 2007), and only a part of these methods has to be serialized and sent to the client, thus making the solution relatively simple and easy to implement.

We implemented the serialization by sending the method name and serialized parameters through Java ObjectOutputStream to the client. Even though not all parameters are serializable as such, they can be serialized by writing custom utility methods for that. In addition for the Graphics2D methods, only two more methods had two be implemented, one for indicating that a new frame begins, and one for indicating that the frame ends and can be flushed to the screen on the client side.

In our solution, on the client side a thread is reading ObjectInputStream for incoming methods and parameters, and calls the local Graphics2D class methods with the parameters read from the stream. Because the Java object serialization format is not optimal for bandwidth usage, the data was compressed with the GZIP algorithm before it was sent through the socket.

The implementation described above was not feasible for displaying complicated graphics as such, but by using a simple cache solution the performance improved to a decent level. The cache was not implemented in Graphics2D interface, but in Simantics paint code. The main idea was to add id for every paintable component, and create a separate Graphics2D object for each paintable component when create method in Graphics2D class is called. The client could now store the paint sequences by component id, and the server side only needs to check whether the component has changed or not, and send

the paint sequence again only if there is changes. This solution requires no changes to Simantics if cache is not used. Even with cache, the changes are small and can be implemented without breaking the Simantics workbench implementation.

By compressing the data and using the cache solution, Graphics2D serialization turned out to be a decent solution. With the simple reference model, the average FPS was between 5 and 10 with 3G connection and between 15-20 with broadband. With these frame rates interactive operations worked smoothly and the usability was generally good, however the performance varied significantly depending on the model. For example the advanced model was not usable with 3G connection. With GPRS or EDGE connection the solution was not usable at all. The frame rate was between 1 to 5 and the system did not feel interactive.

Even though the Graphics2D interface serialization was a working solution, it was far from optimal. The main reason for implementing it was that it required only small changes to Simantics itself, and that it required only small effort to try it out. However, the prototype implementation showed that the thin-client idea for Simantics works, and we decided to implement a far more optimal solution, even though it required major changes to Simantics itself. This improved solution is described in the next section.

4.3 Scene graph

A scene graph is a data structure for presenting graphics in graphical scene. The data structure is independent and thus can be rendered as such, which makes it ideal for remote graphics implementation since the whole data structure can be sent to the remote site where it can be rendered like it would be rendered locally. Scene graphs can also be used to implement collaborative applications by placing the data structure in a shared memory. Even though scene graph is usually a static data structure that can be altered only by adding and removing nodes, it is possible to implement a parametrized scene graph where graphical nodes can change their state according to given parameters. In our case, the scene graph is a tree structure where each node is a Java object. In a traditional scene graph the nodes would be geometrical shapes, but in our case those are renderable Java objects that contains custom render code with support for parameters. (Reitmayr and Schmalstieg, 2005; Inostroza and Lemordant, 2003)

The idea is to create a scene graph based interface for diagram visualization, thus clearly separating paint code from the application logic. This approach allows us to locate the actual paint code on the client side and send only the mandatory parameters for these painters through the network. These painters are independent Java classes that take only serializable objects as parameters and can have multiple painters as children. Both, the server and the client has its own copy of the scene graph data structure, but

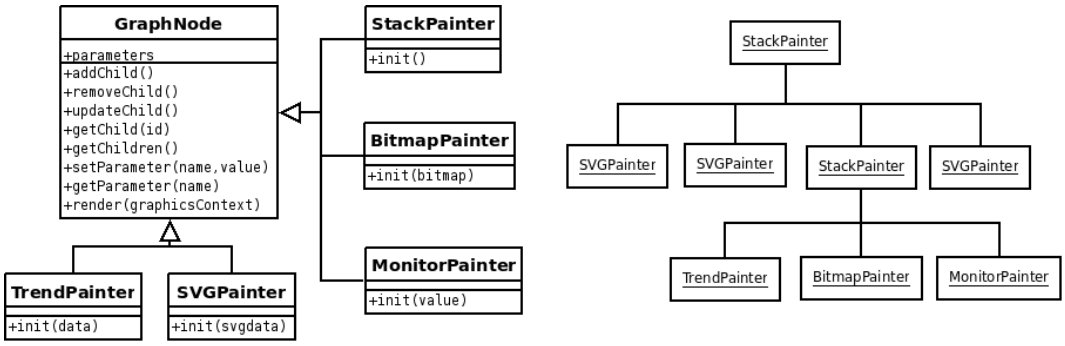


Figure 4.2: Left side shows the class diagram of the scene graph data structure. Each painter class inherits GraphNode. On the right side is described the tree structure of a sample scene graph. The sample data structure contains three SVG images, one trend, one bitmap, and one monitor. The custom BitmapPainter allows us to display any kind of graphics in the scene graph, although bitmap graphics will consume more network capacity than vector graphics.

only the server side can do changes to the structure. After the server has updated the data structure, the changes will be synchronized to the client which can then render the next frame.

Compared to the Graphics2D implementation, the scene graph technique is much more efficient and the benefits are clear: Calling a single painter that performs multiple paint commands requires less operations than calling all the paint operations directly. Figure 4.3 compares these two solutions. In the example we have a basic monitor painter that simply paints a filled and stroked box with text inside it. When the painter code is already in the client side, we only need to send the painter name and the three parameters for it, but in Graphics2D serialization we have to use six methods and transfer 18 parameters.

The scene graph solution requires that the painter code is already located on the client side, but it is not an issue since both, the server and the client can use the same code, hence the painters can be packaged with the client when a new version of the server is deployed. When the client and the server is implemented with Java, we are actually not even limited to use only predefined painters that are located on the client side, but we can dynamically load new painter classes from the server. We will discuss more about this remote class loading later in this chapter.

In our implementation, the server keeps its version of the scene graph data structure on the session specific memory. Simantics then updates this data structure in local memory and the server synchronizes the data structure with the client when requested. By using this approach, the client has exactly the same data structure as the server has, and the client is able to render the graphics for the user at any time.

Graphics2D	Scene graph
"setColor"	"MonitorPainter"
r	x
g	y
b	text
"drawRect"	
x	
y	
width	
height	
"setColor"	
r	
g	
b	
"fillRect"	
x	
y	
width	
height	
"setFont"	
fontName	
"drawString"	
string	
x	
y	

Figure 4.3: A serialized sequence of operations required to paint a simple monitor in Graphics2D and in scene graph. Even though this is a very simple example, the difference is significant.

When a new frame is painted from ten to twenty times in a seconds, only a few items change in the picture between each frame. For example the model configuration is usually static, and only the values in monitors might change. Even when the user navigates on the diagram, the diagram itself does not change, but only the transformation under the diagram changes. We used the cache solution already in the Graphics2D prototype, but in scene graph solution we get even greater advantage of it.

In a scene graph, cache can be easily implemented by storing the scene graph data structure into client memory, and keep track on changed variables on the server side. When the client requests a new frame, the server will send only the changes, not the

whole data structure. The implementation of the cache should be transparent for the programmer, thus the scene graph implementation should automatically compare old and new values of the scene graph data structure and update only changed values to the client. This way the programmer using the scene graph interface does not have to perform any caching or optimization, but the scene graph component does it automatically.

When cache is used, the scene graph data structure is transferred to client side only once, thus most of the data is transferred during the initialization, and the rest of the frames contains only updates to changed variables. Thus, the constant bandwidth requirement is low and interactive operations should be usable also with a slower network connection. With a slow network connection the initialization time is longer, but after initialization the usability should be almost as good as with broadband.

4.4 Input event handling

User interactions are handled as input events on the client side. To interact with the server, these events must be transferred to the server side. Transferring input events from the client to the server does not usually cause problems since one event contains only a small amount of information and events are not sent constantly. However, with a really slow and high latency connection, it makes sense to optimize the traffic from the client to the server also. All keyboard events are usually important and cannot be ignored, but there is a large number of mouse events generated when the user navigates on the Simantics diagram. Some of these mouse events are never used, and thus should not even be transferred to the server side. If we analyze how Simantics handles the events, we can notice following:

- Mouse move events are not needed. The client should send mouse events only when mouse is clicked, dragged, or scrolled.
- Events does not have to be sent to the server immediately, but can be queued until the next frame is requested.
- Simantics has its own event structures that do not require all the information what is in the Java AWT events.

Only by ignoring the mouse move events, we get significant performance improvement. However, we still get quite much events when the mouse is dragged (moving mouse with button pressed). Mouse dragged events are required for browsing the diagram, but only one event for a frame would be enough. By using event queue, we

can ignore old mouse dragged events when a new event comes in, and flush only one mouse dragged event with other events when the next frame is requested. With this logic, more mouse dragged events will be sent on a faster connection, and less on a slower connection because faster connection means better frame rate and the event queue is flushed more often. With this arrangement we can always provide user experience optimized for network speed.

4.5 Remote class loading

As mentioned earlier, all the paint operations are performed by painter classes, and the client application has to have the same painter class implementations what the server has. In case new painters are introduced into Simantics workbench, the same painters must be updated to the client side also. This might be a problem in the future when third parties can implement customer painters. However, Java allows us dynamically load class implementations from the server to the client, and thus display graphics generated by third party painters.

Java supports dynamic class loading from remote locations, hence the client can request missing classes from the server and the server can find the class implementation from the classpath and send the class implementation as bytecode back to the client (Sun Microsystems, 2004). In theory, the server could send the required class implementations beforehand, but in that case the server should be aware of the dependencies what the class has and send also those classes to the client. In practice, going through the whole dependency hierarchy is not feasible, and thus the client must request the missing class implementations on demand. Basically the remote class loading implementation is very straightforward but at least one possible problem can be identified. Because the missing class must be loaded during the scene graph update sequence, we will likely have data coming from the server through the socket stream when the new class is requested. The client cannot proceed processing this incoming data because it waits for the class implementation, and the class implementation cannot be fetched from the server because there is still incoming data for the scene graph. This kind of situation must be identified and the frame data must be read to a buffer to allow the client to receive the class implementation in the middle of the scene graph update sequence. This solution is described in more detail in section 4.6.4.

In remote class loading the different Java versions will cause problems, because classes compiled with one version might not be compatible with other versions of Java virtual machines. Simantics is implemented in Java 1.6 but the applet needs to support Java 1.5, which is the latest Java version for many older but still popular operating system versions. Normally we could compile the painters with Java 1.5 for the applet, but

with remote class loading, the painters comes directly from Simantics codebase and are compiled with Java 1.6. However, we have decided not to support Java 1.5 after the remote class loading is implemented and released.

4.6 Thin-client architecture for Simantics

Previous sections gave background information for the thin-client architecture which is presented in detail in this section.

4.6.1 Key functionalities

Based on the requirements and the background information, following key functionalities can be identified for the thin-client:

- Send and receive objects through network socket.
- Display custom GUI components that will generate events when clicked.
- Send mouse and keyboard events to server.
- Switch to fullscreen and back.
- Fetch whole scene graph data structure from the server.
- Synchronize updated scene graph data structure from the server.
- Load class implementation from the server when class is not found locally.

With these key functionalities, it should be possible to implement all the required features. This is not a very detailed list, but most of the application logic is on the server side, hence the main functionality of the client is to send events and commands to the server and display the results by rendering the updated scene graph.

On the server side, the implementation is based on a full featured Simantics workbench, and thus most of the application logic is out of the scope of this thesis. However, we will focus on developing a software component that communicates with the client and with Simantics, and thus acts as a wrapper between these two sites.

The server component must be able to perform at minimum following functionalities:

- Send and receive objects through network socket.

- Send full scene graph data structure to the client when requested.
- Synchronize updated scene graph data structure with the client.
- Change active diagram when client requests.
- Start and stop experiment.
- Provide class implementations for the client when requested.

For most of the functionalities in the server side there is a corresponding client-side functionality. This makes sense since the server software is a wrapper between the client and Simantics. The class diagram in the figure 4.5 gives us a better picture of the role of the server side implementation.

In addition to the run-time functionalities described above, the server must perform following actions when a new client connection is accepted:

- Initialize Simantics platform.
- Connection to Simantics database.
- Load requested project from the database and initialize the project specific plug-ins.
- Load the requested model.
- Load and start the requested experiment.
- Load the requested diagram presentation of the model.
- Create scene graph presentation of the diagram.

4.6.2 Class-level design

Client-side class diagram is presented in figure 4.4. As shown in the figure, we have a main class called ThinClient and an applet wrapper class SimanticsApplet. Communication with the server is performed by the CommunicationChannel class which is used by the ThinClient and also by the RemoteGraphicsClient. The RemoteGraphicsClient is the class that loads SceneGraph data structure from the server and renders it for the user. In addition to the SceneGraph, the ThinClient can have multiple client side GUI components, such as buttons and text fields.

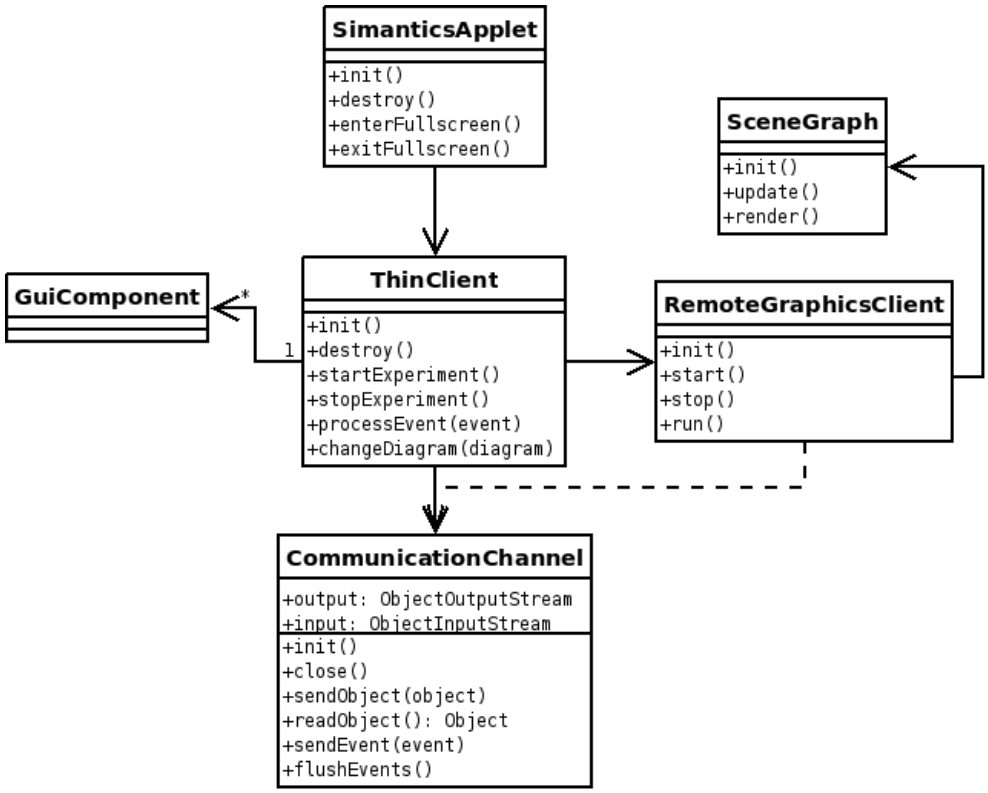


Figure 4.4: High-level class diagram of the thin-client. ThinClient is the main class that displays the graphics and handles events. In addition to the RemoteGraphicsClient that displays the graphics from the server, there can be custom GUI components on the client side. These components can be used for example as shortcuts for keyboard events.

On the server side, we have a ServerNode class that integrates Simantics Platform with the thin-client through the CommunicationChannel as shown in figure 4.5. ServerNode is responsible for performing operations in Simantics Platform, and the Simantics Platform will update the SceneGraph data structure independently. This data structure is then synchronized with the client by the RemoteGraphicsClient through the CommunicationChannel.

4.6.3 Frame update sequence

Figure 4.6 presents high-level view of initialization sequence. As shown in the figure, the initialization consists of three steps: loading the simulation model, loading the experiment, and loading the requested diagram presentation for the model. A model can have multiple experiments and diagrams, hence model is loaded once in the beginning,

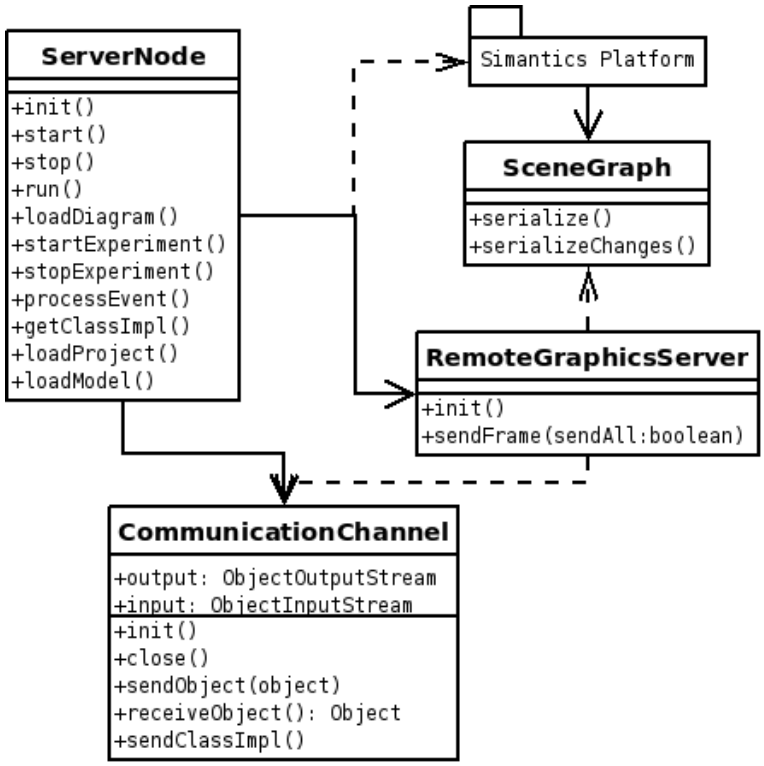


Figure 4.5: High-level class diagram of the server. In this diagram we assume that only one client is served at a time.

but experiments and diagrams can be changed during the session. Requesting the first frame differ from other frames, hence the client must request a whole data structure when the diagram is changed. It is also important to notice that the next frame is requested immediately after the client has received first message of the previous frame. With this way, we can optimize frame rate and latency. The frame updates must be synchronized because we have an interactive application where user must get feedback from the performed events as soon as possible, thus the server should not push new frames to the client before it has received and handled the events from the client.

A detailed view of a single frame update sequence is presented in the figure 4.7. The basic functionality of the thin-client is to load the scene graph from the server, display it for the user, get input events from the user and send them back to the server. After this, the server performs required operations and synchronizes the updated scene graph data structure with the client.

In the next section we will describe the communication protocol used for the communication between the sites.

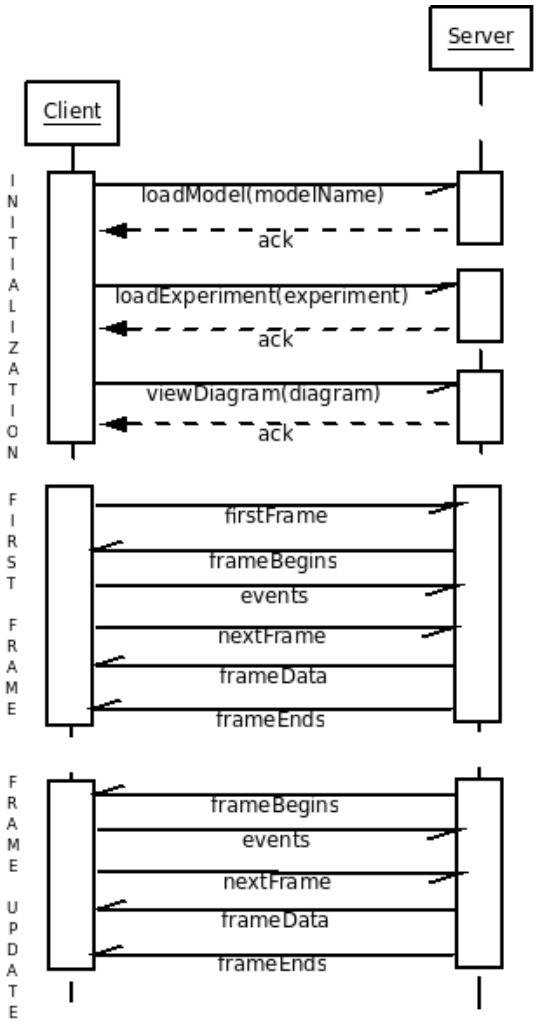


Figure 4.6: Sequence diagram describing the initialization of the connection and the first two frames. Notice that the following frame is requested before the previous frame is even read. Using this practice the server can prepare and send the next frame when the client is still reading the previous frame, and thus we get better frame rate.

4.6.4 Communication protocol

Java provides object based communication streams that can be used to send and receive Java objects through a network socket. For reading objects from the stream Java has ObjectInputStream, and correspondingly ObjectOutputStream for writing objects to the stream. We use these two streams to send and receive data on the both sides. In addition, the data serialized by the ObjectOutputStream will be compressed with GZIP

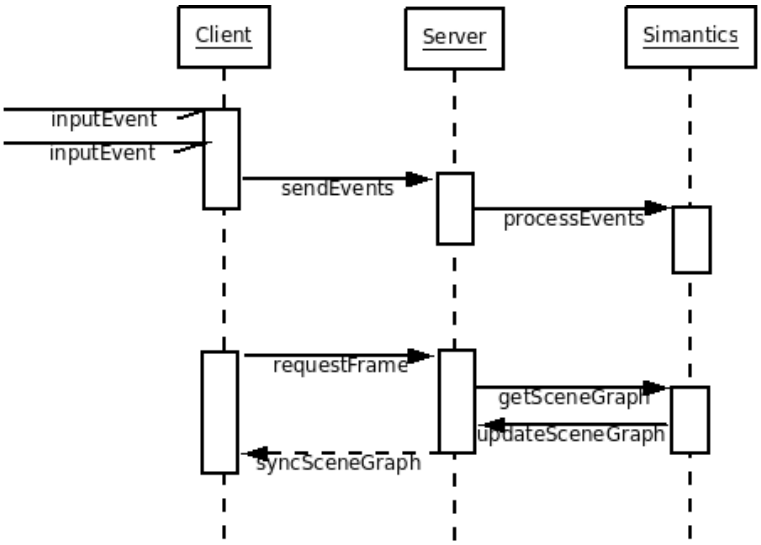


Figure 4.7: Sequence diagram describing the frame update sequence. Event handling is asynchronous and the client queues events before they are sent to the server. Usually events are sent to the server right before the next frame is requested, but on a slow network connection they can be sent between the frames because the frames are requested infrequently.

before it is sent, and correspondingly data from the stream will be decompressed before it is forwarded to the ObjectInputStream.

Communication from the client to the server consists of input events and control commands, such as start experiment, stop experiment, load diagram, and request next frame. All control commands are Java String objects and all events are custom objects. Communication from the server to the client consists of acknowledgements and scene graph synchronization data, meaning painter name and parameter sequences.

The remote class loading described earlier causes problems with the communication protocol, since a new class will most probably have to be requested when a frame is not yet completely read from the stream. In this situation, the client should read all incoming data from the stream to a buffer before requesting a class implementation. After the class has been fetched, the client can continue processing the frame from the buffer. However, reading the stream data to a buffer should not be the default operation, since it consumes memory and causes performance loss. The figure 4.8 shows how the incoming scene graph data is buffered and the remote class loader is changed to listen to the socket. When the class implementation has been fetched, the scene graph component is switched back to listen the socket. Switching between these two states does not cause significant performance loss since the client does not have to fetch missing class implementations that often, and most of the remote class loading

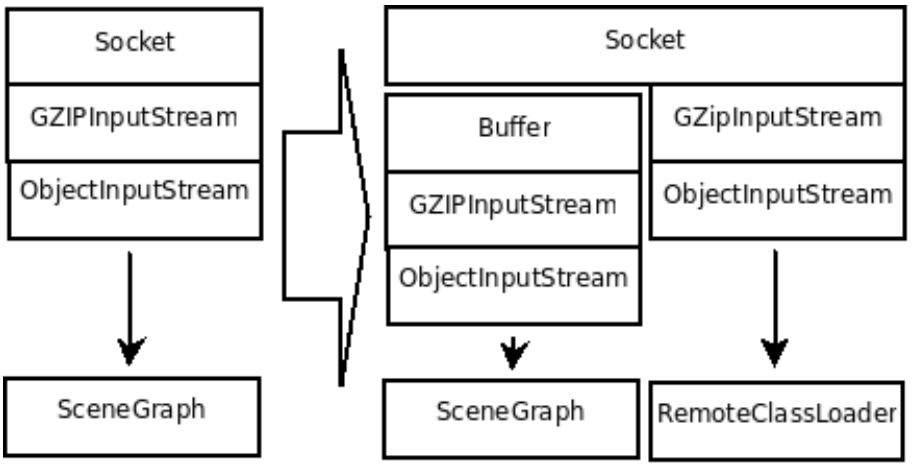


Figure 4.8: The input stream is buffered when a new class implementation is requested from the server, and the data from the socket is directed to the remote class loader.

requests are performed during the first frame.

4.7 Implementing the thin-client

This chapter described the architecture for the thin-client implementation of Simantics. The architecture contains two main components, the thin-client application and the server side Simantics wrapper. The main problem in designing the architecture was to solve how the graphics could be efficiently transferred from the server to the client. We also took into account the support for the possible third-party plug-ins for Simantics by designing remote class loading mechanism.

The components designed in this chapter will be implemented and used to measure the performance of the architecture. We do know that the current release of Simantics is not stable, and thus we do not aim to make a perfect implementation yet, but a working solution that can be implemented further after we know whether or not the designed architecture is suitable for the system.

In this chapter we assumed that we have one client and one server in our system. In the next chapter, we will discuss how we can keep the quality of service high even if the have a number of concurrent users.

Chapter 5

Server-side scalability

In the previous chapter we described the thin-client architecture for the Simantics simulation environment with the assumption that we have only one client and only one server. In reality we must be able to serve multiple concurrent clients, and scale up to thousands of concurrent users. Like we have noticed earlier, some simulators are lightweight and does not cause performance issues, but there are simulators that perform resource intensive calculations, and thus might require large amount of computing power. It is likely that one physical machine does not have enough resources for running the required amount of concurrent simulations, hence server-side distribution is required to make the system scalable. In addition, at some point the network bandwidth on the server side can become a bottleneck, and thus we must design the architecture as such that we are able to serve the clients from multiple network locations. This chapter describes how the server-side implementation can be designed to support multiple concurrent users, and how the system can be made to scale up somewhat linearly by increasing the amount of physical servers.

5.1 Quality of service

The thin-client application we are developing is a part of software that is provided as a service. Because we expect to get also paying customers for our service, we have to provide commercial quality, thus the quality of service must be high. The main quality related non-functional properties are performance, dependability, security, and safety (Barbacci et al., 1995). Performance consists of attributes such as latency, throughput, and capacity. Throughput and capacity can be increased in our system by increasing the amount of servers. To minimize latency, we can only minimize the delays caused by our software components, but we cannot do much for the delays caused by the network.

The most important dependability attributes are availability and reliability, which both can be maximized by duplicating components. To make a distributed system secure is such a demanding task that we do not pay attention on the security issues in this thesis. Safety, however, is currently not an issue in the system we are designing. This chapter described how these quality attributes can be taken into account while the system is made scalable. (Coulouris et al., 2005)

By distributing the work load of multiple concurrent clients to multiple servers, we ensure that the performance of the system stays good enough even if the amount of users increases. In addition, to increase the reliability of the system, we will duplicate every component on the server side, and thus on hardware failure we are able to replace the broken hardware with a working one. However, this does not mean that we will have to have spare machines for each server, but we can have for example two servers actively performing the same tasks, and on a hardware failure the traffic is directed only to the working one. The users can still be served, even though the performance might not be as good as with two servers, but the situation is just temporary.

5.2 Serving multiple concurrent clients

The basic solution for serving multiple clients is to deploy multiple server processes, one for each client. However, this solution is not very efficient since it would require loading Simantics platform in to memory multiple times. The platform has large amount of components that can be shared with multiple sessions, and thus one Simantics instance should serve multiple clients concurrently. However, we still need multiple server processes because we have different Simantics configurations and multiple databases that should be accessed. This means that one process will serve multiple clients by using threads, and one server machine will have multiple processes running concurrently.

If we look at the initialization procedure of a single connection described in the previous chapter, we can notice that there are steps that can be performed only once for a process. These steps are actually the most time consuming steps, and thus the initialization time of a single client connection will get significantly shorter when these steps are not performed for every connection, but only once when the process is started:

- Initialize Simantics platform.
- Take a new connection to the database.
- Load the project from the database and initialize the project specific plug-in components.

- Load the requested model.
- Load and start the requested experiment.
- Load the requested diagram presentation of the model.
- Create scene graph presentation of the diagram.

Because connecting to database, loading a project, and initializing the required plug-in components takes time, these should not be performed for every client session, but should be shared with multiple sessions. This requires that we have own process for each database and project, but each of these processes can serve all models belonging that project. When the process starts, it connects to a database and loads the project configuration from the database. After the project configuration is loaded, the required plug-in components are identified and initialized. The model configuration is loaded and initialized from the database when a client connection is accepted. If the model configuration is updated in the database while the server is running, the server should synchronize its copy of the project and model configuration. However, Simantics has built-in support for synchronizing the database on the fly.

By serving multiple clients with a single server process, we save system resources since most of the Simantics software components inside the server process will be shared with the open models, and in this case with the connected clients. When a new client connects to the server, it will cause only a small increase in memory and CPU consumption. However, when a single process serves multiple clients, possible errors or thread crashes on one client might reflect to other clients on the same server. The other users might suffer for example from performance issues or the other connections might even be dropped. For this reason we must pay extra attention on error handling.

In addition to the servers running concurrently with different configurations, we might need to duplicate the most popular configurations. By duplicating some processes into two or more physical server machines we can serve more clients and the reliability and the availability of the system increases. When we have multiple identical locations which we can use to serve the client, we have to develop a load balancing component that directs the traffic to the server that has most free resources, and thus balances the load between the servers.

5.3 Transparency

One requirement for the system is that the distribution is transparent, thus all connections from the clients to the servers go through a single IP address and the location

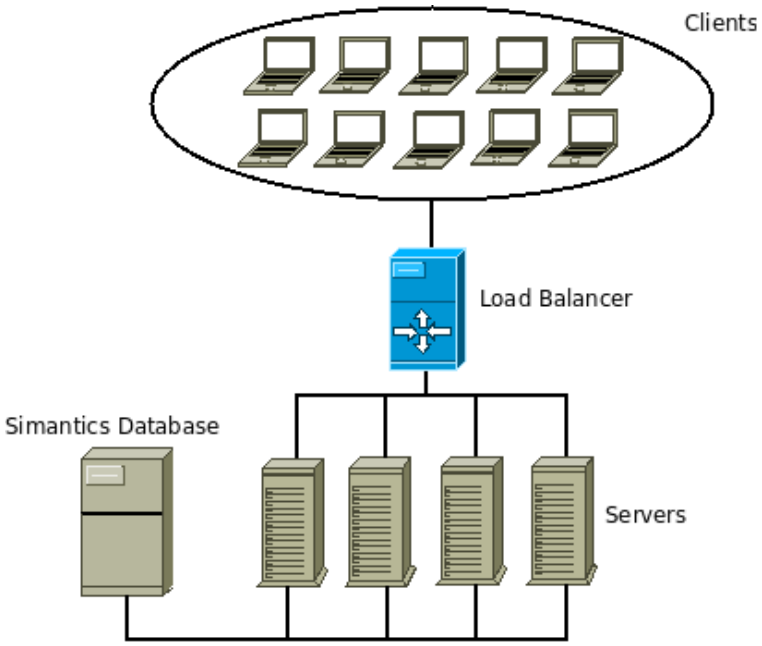


Figure 5.1: High-level architecture when using single load balancer. All client connections go through the single load balancer where they are directed to different backends.

of the server serving the user is never shown. Transparency is required because most computers have firewalls and those need to be configured to allow connection to a specific address. If the architecture would not be transparent, the user would need to configure the firewall every time when a new connection is established. Thus, we want to minimize the need for extra configuration and serve all clients through a single address, hence all the traffic must go through a single load balancer.

The easiest way to make the solution transparent, is to direct all connections through a single load balancer as shown in the figure 5.1. The reliability and scalability of this architecture is a problem, because all traffic goes through one point, which in this case is the load balancer. However, the figure shows only the machines that are active concurrently. We can still have another load balancer as a spare that activates only if the main load balancer fails. However, this solution does not provide reliability on network failure.

With a single load balancer we can make the distribution architecture transparent for the user, but in that case we are dependent on a single network connection and capabilities of a single physical machine and software. If we increase the amount of load balancers, the server side distribution will become visible for the user, and thus we lose the transparency which was one of the requirements for the architecture. In addition, the heterogeneous server processes and shared sessions are problem when we have

multiple load balancers visible for the end user, since each load balancer usually has its own set of backend servers in use, thus the load balancer might have to be chosen according to the content that the user wants to access. However, by being able to locate the servers to different sites, the system would scale up almost unlimitedly.

When the number of concurrent users becomes high enough, single load balancer is not capable for handling the traffic, thus multiple load balancers are required. The simple solution for having multiple load balancers would be to use Domain Name System (DNS) based load balancing that uses simple round robin algorithm to choose the address and thus the site from where the client is served (Brisco, 1995). However, because load balancing should not be visible for the user, except in form of better performance and availability, we cannot randomly direct the users to different load balancers. One approach for hiding the distribution from the user is to locate the load balancers to different locations, but to direct the same user each time to the same location. Because the thin-client is inside our web site, it is possible to use different user identification technologies to identify the user and make sure the user gets the same address every time. This can be done for example by storing the client IP address into the site database, or by calculating hash value from the client IP address and selecting the backend corresponding the hash value. However, for example computers on a DHCP network or mobile devices can have dynamic IP address that can vary, and thus the we cannot identify the user by the IP address. In these cases we can, however, identify the user by using web technologies such as cookies or username and password identification.

The solutions described above assumes that all sites are able to provide all the content in the system. If we would distribute our content across the sites so that one site is capable for serving only a part of the content, we would lose the transparency, but would get better scalability. This kind of solution could be easily implemented by directing the user to the site which serves the requested content. This solution would require us to use a sophisticated method for distributing the content for example so that same kind of models would be server by same site, and thus the possibility that the same user would be server by the same site would increase. Hence, the need for configuring firewall would decrease.

Figure 5.2 shows the high-level architecture for multiple load balancers. Although the clients are connected to the system through different load balancers (and addresses), this does not have to be visible for the client. If the same client is connected to the system always through the same load balancer, the client does not see that our system actually has multiple load balancers serving the large customer base.

In any case, one publicly visible server must serve more clients than one commodity server machine can serve. To solve this problem we use load balancer component that accepts connections on a public IP address. This load balancer will distribute the load

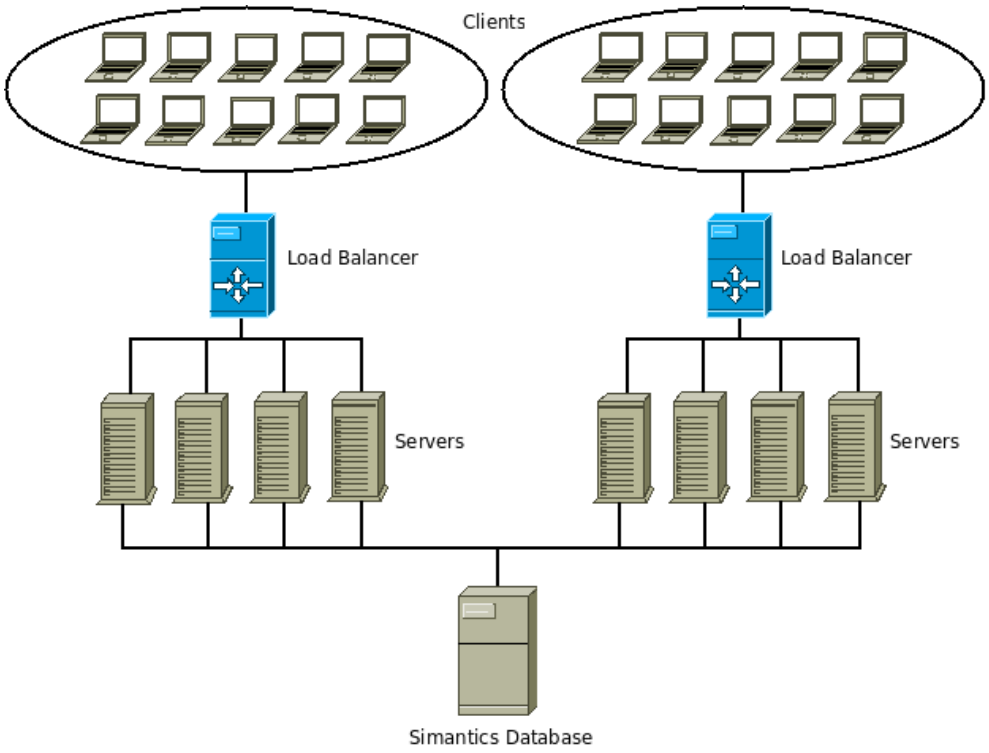


Figure 5.2: High-level architecture when using multiple load balancers. Notice that there is no connection between the load balancers, and different clients accesses to different load balancers. In the picture we have only one Simantics database, but there could be more and they does not have to be shared with the load balancers.

across the backend servers by directing the incoming client connection to the server that has the lowest load, and thus can provide best service for the user. Next section describes the load balancing in detail.

5.4 Challenges with load balancing

Because Simantics is an extensible software, it can be configured for different purposes using plug-ins, thus one configuration is not capable for serving all simulation models. In Simupedia we want to provide different simulators for the users, and thus we need servers with different plug-in configurations. This means that the backend processes are not similar but heterogeneous, and the load balancer must be aware which servers can serve which models.

To handle heterogeneous process environment, we can arrange the backend servers to

groups. Members of each group are similar, thus can serve same models. When a new connection comes in to the load balancer, the load balancer first selects the correct group for the client, and then searches the backends for this group. The load balancing algorithm is then performed on the servers in this group and the client if redirected to the server that has the lowest load, thus has the best capabilities to serve the client. However, it is likely that we locate multiple server processes serving multiple groups on a single physical server, we must have a method to measure the load that all these processes causes to the server.

The system is required to allow dedicating computing power for a specific user group. The usual case is that the customer has bought a simulation environment for a specific amount of concurrent users, and we must be able to provide enough computing power for that group around the clock. This feature can be easily implemented by using the grouping described above. If we simply put all models from the customer to a same group, and locate the server processes for that group to a dedicated physical server, we have a dedicated server for the customer. We do not need any special handling in the load balancing algorithm for this kind of situations.

In Simupedia concept, we have simulation models that can be played as a multi-player game. In these games each player must share the experiment session, and thus must be served by the same server process. For this reason, the load balancer must have a lookup table of active experiments and when a new client requests access to an existing experiment, the connection is immediately directed to the backend process where the experiment is running. No load balancing strategies will be applied in this case. Although this will cause unequal distribution, it will not cause significant problems since only a small number of players will be attached on a single game.

The main problem with shared sessions is how the load balancer gets the list of currently running experiments. Another problem is related in performance since we must keep additional lookup table for the experiments and for each connection we must perform lookup by experiment identifier. This feature also requires that the client can tell into which experiment it should be attached to already when the connection is initialized, thus the experiment identifier must be already in the connection string and the user cannot change the experiment during the session.

5.5 Load balancing algorithm

Load balancing is widely used technology for partitioning computing tasks according to load distribution strategies to utilize available computing resources effectively. Load balancing is a key technique to improve scalability of network software system. On our system, we want the load balancer to distribute incoming connections evenly for

each backend server. A backend server will serve the user for the whole session, and each backend server should be equally loaded. Depending on the system, the load measured can be CPU load, memory load, or network load for example. In our case, we are interested in CPU and memory load, since we have managed to minimize the network traffic in our thin-client architecture. (Youn, 2005)

By taking into account the constraints and requirements presented in the previous section, we can design the final load balancing algorithm for our system. This algorithm is a modified weighted least connections (WLC) and the main procedure is described below.

- If there is no group capable of serving the model, thus the model does not exist, the connection is denied.
- If the user wants to attach to an existing experiment, the connection is directed to the server where the experiment is running.
- On other cases, the WLC algorithm will be performed for the servers belonging to the group that is capable for serving the model.
- If the host specific WLC will give multiple choices with same weight, the process with least connections is chosen.

In WLC, a weight is calculated for each backend process separately by using formula W = Ci/Wi, where Ci is the number of concurrent connections to the physical server where the process is running on, and Wi is the weight of the physical server where the process is on. Notice that if we have two processes in one physical server, they will both get the same weight, even if they would not serve same number of concurrent connections on that time. The idea of this algorithm is to distribute the load equally between physical servers, hence it does not matter if the load is not evenly distributed between the processes inside the server. However, if we have multiple instances of the same process on a server, the load does not distribute equally between them. The solution is to take the process specific connections into account when there is more than one process on a single server. This calculation can be performed after the WLC algorithm because WLC will give same weight for these processes, and thus we can simply run another check for the processes that has the same weights after performing the WLC.

The algorithm described above is relatively simple but efficient and thus causes minimum delays for session initialization. Even though the load is estimated from the number of concurrent connections, the load is likely to distribute somewhat evenly across the servers. Although the resource consumption varies between the sessions, we can assume that the resource intensive sessions and the sessions that consumes

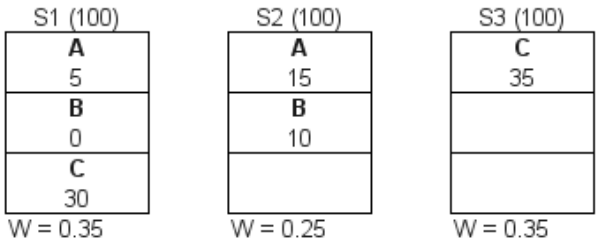


Figure 5.3: In this example, all server have the same weight (100), and if the load balancer accepts a new connection to group A, it would direct the connection to the server S2, since it has the lowest value, even if the process in the server S1 would have less connections.

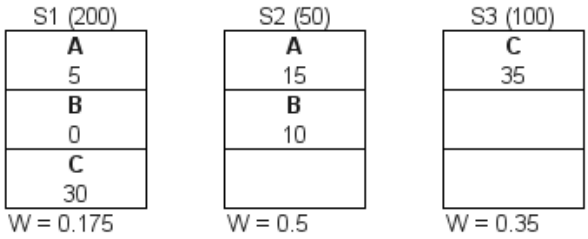


Figure 5.4: In this example a connection to group A would be directed to the server S1, since its value is lowest even though it has most connections. In this situation S1 weight is higher than S2 and S3, meaning that is has more computing power than the two other servers, hence it can easily accept more connections.

only slightly resources are accepted, and thus distributed randomly across the servers. For this reason, it is very likely that a single host will serve both, highly resource intensive and less resource intensive sessions. Hence, the connection count gives a good estimate of the server load.

Figures 5.3 and 5.4 exemplify how the algorithm works in two different situations. WLC values are server specific, hence each process in the same server gets the same value. The incoming connection will be directed to the server with lowest value. In both examples, we have three servers (S1, S2, S3) and three process groups (A, B, C). The weight for each server is reported next to the server name, and the active connections for each process is reported under the group name on each server. The calculated value for each server is reported under the server. This value is the same for each process running on the server.

5.6 Implementing the load balancer

To make the solution scalable we need to implement the load balancer component, thus in the final system we have three components: the thin-client, the server implementation and the load balancer.

The load balancer must be aware of the different server processes running on the system and it must monitor their statuses to be able to redirect the traffic away from the failed hardware. For this reason the load balancer must have database containing information about the server processes and also efficiently implemented load balancing algorithm. Because all the traffic goes through the load balancer, it is a bottleneck in the architecture if it is not designed and implemented well. In addition, its failure would cause the whole system to be out of use. For this reason a spare load balancer should be always online and ready to take the control if the main load balancer fails.

The performance of the load balancer implementation is measured in the next chapter, but we do not expect to see any problems with the performance of the load balancer because we do not have enough server hardware to test the load balancer under very high load.

Chapter 6

Evaluation of the architecture

In this chapter, the architecture presented in chapters 4 and 5 is analyzed against the requirements specified in chapter 3. All the main components designed in this thesis were implemented. The implementations are only preliminary, but we did not even expect to get the final versions implemented during this thesis project. The Simupedia project will continue after this thesis and we will improve the design and implementation of the components according to the conclusions made in this thesis. However, these implemented components are working and contains all the main features, thus we can do the preliminary measurements with them. The results of the measurements provide good background information for the architectural evaluation. After the measurements, the architecture will be analyzed and evaluated using Architecture Tradeoff Analysis Method (ATAM).

6.1 Measurements

All the main components were implemented during the project: the server, the client, and the load balancer. With these components we can measure the performance and the scalability of the designed architecture. However, these components are not final versions, and more optimization is required before they can be taken into production, hence the results are preliminary and must be analyzed in qualitative way. It also must be notices that not only the thin-client components are unfinished, but the Simantics simulation environment is also unstable and contains many flaws.

6.1.1 Test configuration

For the measurements, we have three computers to be used as servers, two computers to be used as high performance clients, and one netbook as minimum configuration client device. The configuration of each device is described below. For each device, the table contains information about operating system (OS), processor (CPU), and memory (RAM). In addition, for the client devices that perform graphical operations, the type of graphics processing unit (GPU) is reported.

Server 1 & 2:

OS	Ubuntu 9.04 64bit
CPU	AMD Phenom 9550 quad core 2.2GHz
RAM	4GB DDR2
GPU	GeForce 8200

Load balancer:

OS	Ubuntu 9.04 64bit
CPU	Intel Core2 Quad Q8200 2.33GHz
RAM	8GB DDR2

Netbook:

Device	ACER Aspire One
OS	Ubuntu 9.04 NetBook Remix
CPU	Intel Atom N270 1.6GHz
RAM	512MB DDR2
GPU	Intel GMA 950

PC 1:

OS	Ubuntu 9.04 64bit
CPU	AMD Phenom 9150e quad core 1.8GHz
RAM	4GB DDR2
GPU	Radeon HD 3200

PC 2:

OS	Ubuntu 9.04 64bit
CPU	AMD Phenom 8650 triple core 2.3GHz
RAM	4GB DDR2
GPU	GeForce 8200

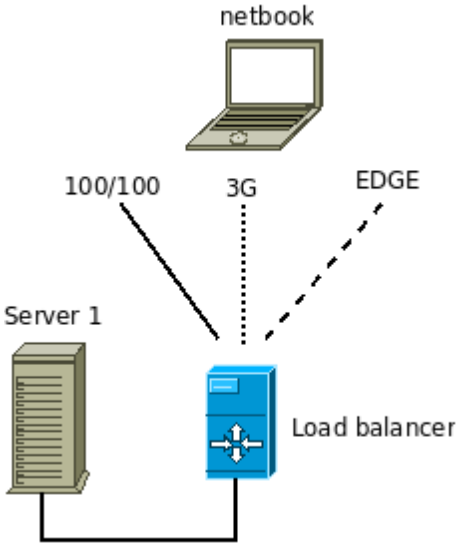


Figure 6.1: Thin-client test configuration. Three types of clients connections are tested: 100/100Mbps ethernet, good quality 3G connection with 1Mbps, and a good quality EDGE connection with theoretical 384kbps. The load balancer and the server is connected with a 1000/1000Mbps ethernet cable.

For the thin-client performance tests, we used one server computer, the load balancer, and the netbook. We used three different network connections for the measurements: 100/100Mbps ethernet, 1Mbps 3G connection, and we also tested the system with a good quality EDGE connection (384kbps in theory). Each connection was tested three times to make sure that there is no significant variances between the results. The results reported in section 6.1.2 are from the second test run, but each test run was very similar to others. Before each test set, the server software and the client web browser was restarted to avoid performance loss caused by for example memory leaks. The thin-client test configuration is presented in figure 6.1.

We developed a measurement tool for the thin-client performance tests. The tool is an applet that does everything what the real thin-client applet does, but displays statistics of the measurements instead of the diagram, thus do not perform the actual rendering. The rendering is passed because it would cause major performance loss in the test computer, and thus we would need more test hardware. However, this should not affect to the test results because in the real applet the rendering is performed asynchronously, hence it does not affect to frame rate, timing, or performance. The test program will measure input and output bandwidth and frame rate, it also displays the time when the first frame is displayed to indicate the load time of the diagram.

To test the scalability of the architecture, we used two backend servers (Server 1 & 2), the load balancer, and two client computers (PC 1 & 2) that are capable for running

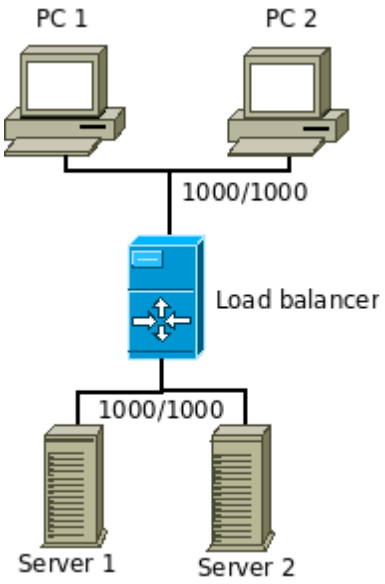


Figure 6.2: Scalability test configuration

100 concurrent client instances both. Because both servers are identical, we will use same weight for the both servers in the load balancing algorithm. We will run one server process on each server computer, and one test program process on each client computer. The test program will not start the 100 connections immediately, but starts one connection each 5 second, thus it takes 500 seconds to get all the 100 connections running. With this procedure, we make sure the Simantics has enough time to initialize the session view before a new connection comes in. This test configuration is described in figure 6.2.

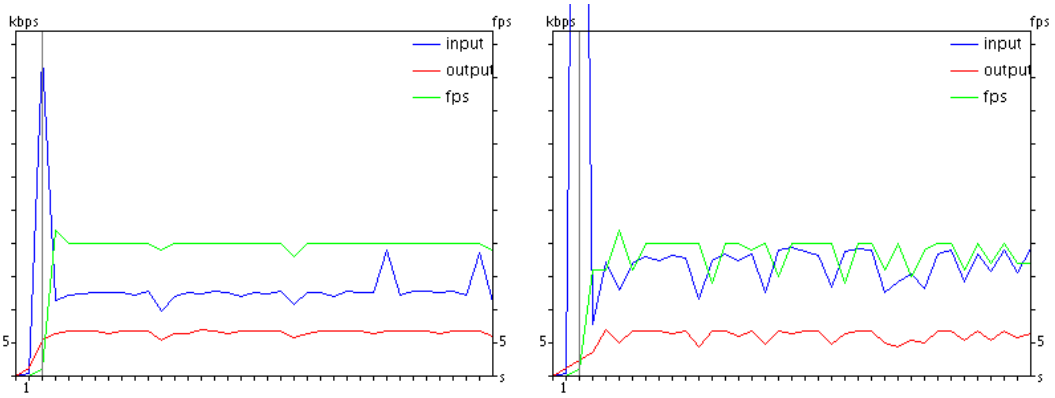
The test program used in scalability tests is a headless client application that do not perform any rendering because we are interested in the server performance, not the client performance. Otherwise the test program is like the real thin-client applet, and it generates transformation events during the test to simulate user interaction. The scalability test program will measure minimum, maximum, and average frame rate. It generates a graph containing these values and the amount of concurrent connections. The system can be said to be scalable enough, if the frame rate is feasible and the application is usable after 100 concurrent connections from each client computer. The usability of the application can be determined by taking one additional client connection with the real thin-client application and running the simulation with it.

In addition to the configurations defined above, we have one Simantics database server serving all our backend servers. With these tests, the performance of the database server is not a problem, hence its existence can be ignored. Also, each test was per-

formed with the both simulation models defined in section 3.4.

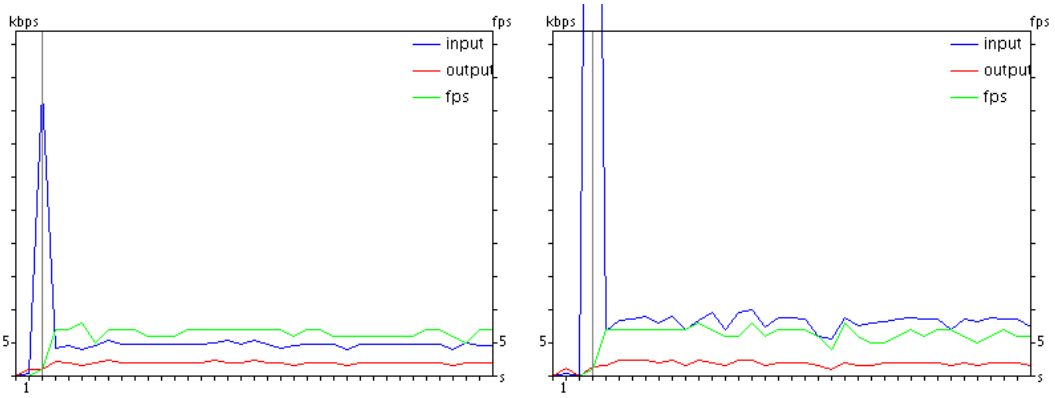
6.1.2 Thin-client performance

The thin-client performance was measured with both reference models and with three network connections: 100/100Mbps ethernet, 1Mbps 3G, and EDGE. Figures 6.3 - **??** shows the results of these measurements. Each graph contains input and output bitrate, and the frame rate that reflects the usability. In addition, the graphs shows when the first frame is displayed for the user, thus it reflects the initialization time of the model.



(a) Simple model **(b)** Advanced model

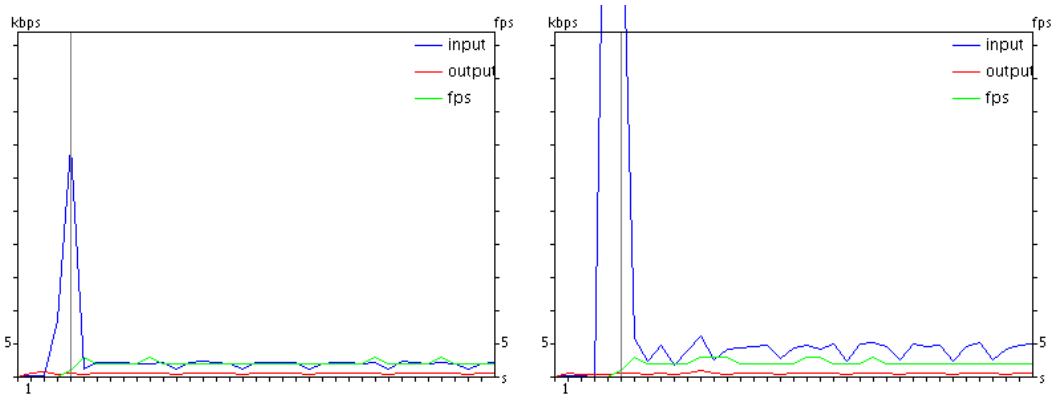
Figure 6.3: Test results using wired 100/100 connection.



(a) Simple model **(b)** Advanced model

Figure 6.4: Test results using 1Mbps 3G connection.

In the results, figure 6.3a shows the simple model with ethernet connection. The average frame rate is the limited 20 frames per second. The limit is set to 20 because better



(a) Simple model **(b)** Advanced model

Figure 6.5: Test results using EDGE connection.

frame rate would not improve user experience, and thus would be waist of resources. The average input traffic is 12kbps and the average output traffic is about 6kbps, and the first frame is loaded in two seconds. The results are promising since the model should work on almost any network connection.

The advanced model with ethernet connection works almost as well as the simple model as shown in figure 6.3b, but the frame rate and the average input bitrate varies significantly, and thus the usability is not as good as with the simple model. However, these variations are not caused by the thin-client implementation or the network, but by flaws in the simulation solver which in this case is the custom DEVS implementation. The solver consumes that much resources that the server process is not capable for synchronizing the scene graph with the client smoothly. Another difference with the simple model is the bandwidth consumption during the initialization caused by the additional SVG graphics on the diagram. However, this increased network traffic does not show for the user because the bandwidth is wide enough.

The results in the figure 6.4a differs already significantly from the measurements made with the ethernet connection. The figure shows the results of the measurements with the simple model using 3G connection. Interesting is that the initialization time is exactly the same with the ethernet connection, however, the average frame rate is significantly lower. As the figure shows, the average frame rate is about 7 frames per second while the average input traffic is 5kbps and the output traffic is 2kbps. These results are significantly lower what the network bandwidth gives us to expect. This behavior can be explained by the synchronized scene graph update process where the server does not send the frame update before the client requests for it. Because the latencies with the 3G connection are significantly longer than with the ethernet, there is longer interval between the update sequences, and thus the frame rate is significantly

lower.

Figure 6.4b shows the results of the measurements made with the advanced model and 3G connection. This graph shows only a slightly longer initialization time than with the simple model. The average input traffic is higher than with the simple model which is understandable, but what is surprising is that the average frame rate is the same or even higher than with the simple model. This difference is probably caused by the 3G and TCP protocol because the latency seems to decrease when the traffic increases.

Figures 6.5a and 6.5b shows the results with EDGE connection. We do not expect very good usability with EDGE connection, but it is interesting to see how well the system works with the minimum network connection. The longer initialization time shows that we do have slower network connection than in the 3G measurements. The average frame rate is about 2 and the usability is poor. We could improve the frame rate by pushing new frames to the client without waiting any reply from the client. However, this would not improve the latencies of the user interaction. The differences with the simple model and with the advanced model are minor.

6.1.3 Scalability

In the scalability test, we used two clients to take connections to the servers and measure frame rate. In addition, we took one client connection manually to evaluate whether or not the system is usable after 200 concurrent connections. The results of the automated test programs are presented in figures 6.6, 6.7.

Figure 6.6 presents results of tests with the simple model. In this test we had two client programs in two client machines, two servers and two server processes in each, thus two client processes and four server processes in total. When the amount of concurrent connections goes up, the average frame rate goes down. The figure shows also the maximum frame rate which is higher than the average frame rate, meaning that there is variation on the performance between the clients. The connection initialization seems to cause significant delays when the amount of concurrent connections increases, and thus it takes time to achieve the full frame rate. Hence, there are no major variances with the frame rates but the new connections causes the average frame rate to be lower than the maximum. This is a positive discovery because it means that the server provides equal user experience for every client after the initialization is fully carried out.

We expected to be able to serve 100 concurrent clients with a single server, and thus the two server should have been able to serve 200 clients. Like the figure shows, we did not achieve this goal because the frame rate shows that the system is not usable after 40 client connections to a single physical server and 20 connections to a single process.

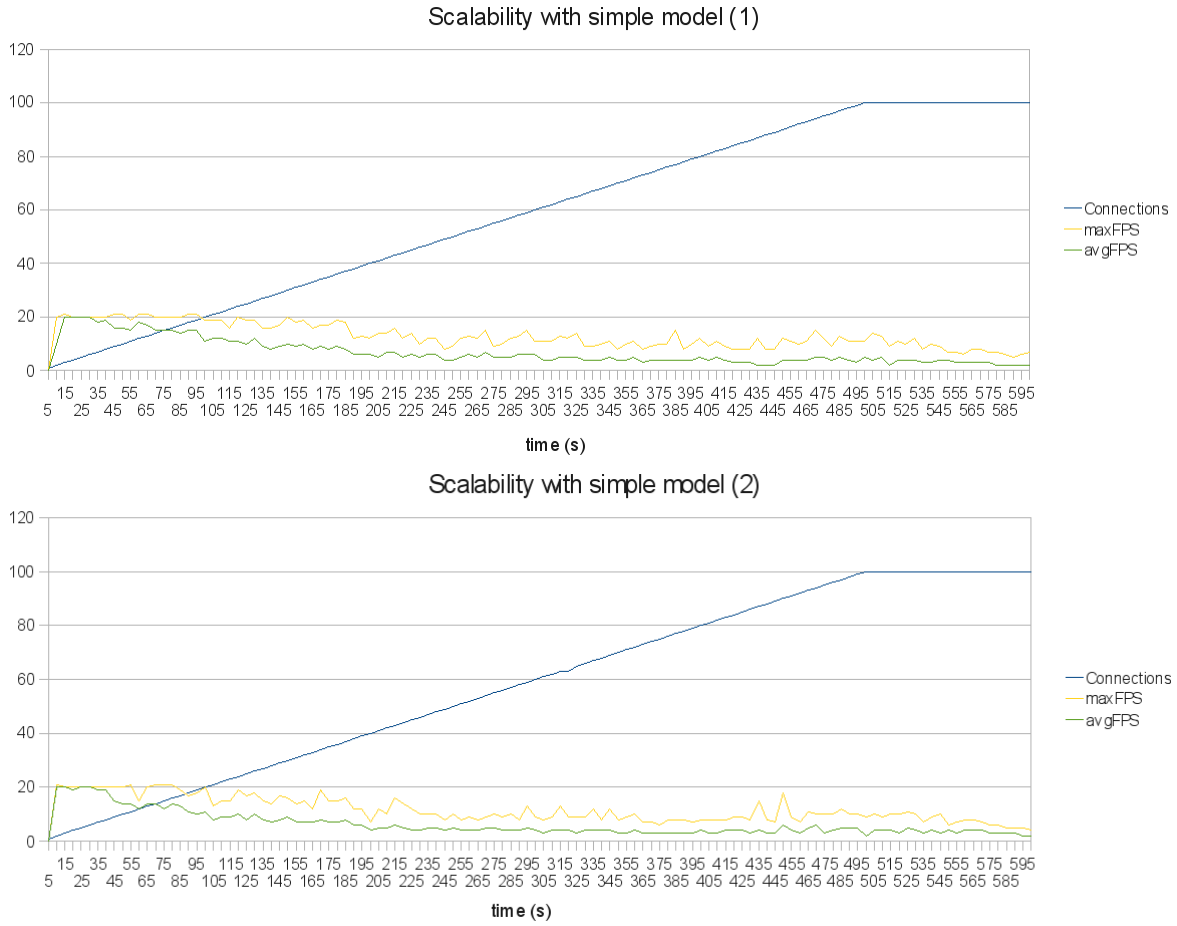


Figure 6.6: Scalability test results using two clients accessing simple model. Because we had two separate client programs in the tests, we have two different graphs. But as the graphs shows, there is only minor differences between the clients. The test configuration contains two servers and two server processes in each.

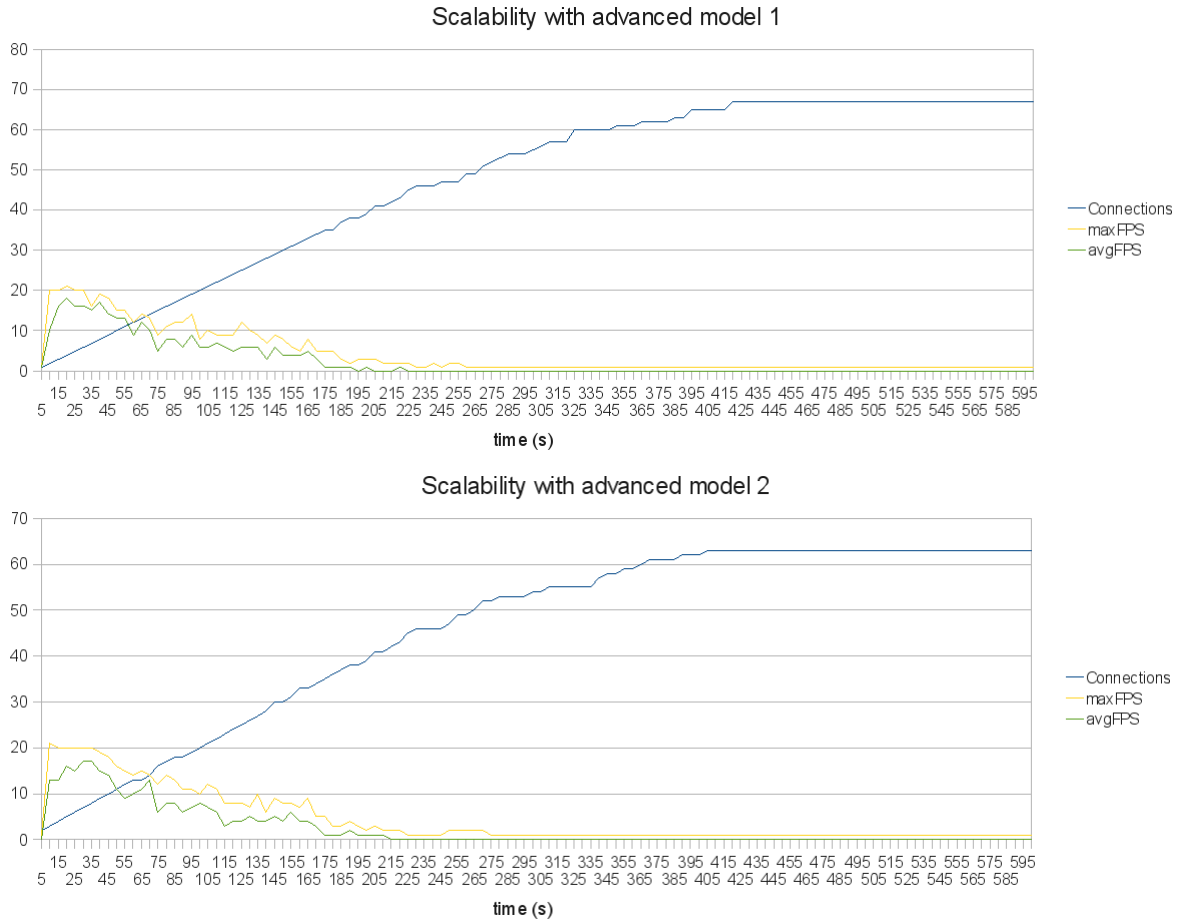


Figure 6.7: Scalability test results using two clients accessing advanced model. Because we had two separate client programs in the tests, we have two different graphs. But as the graphs shows, there is only minor differences between the clients. The test configuration contains two servers and two server processes in each.

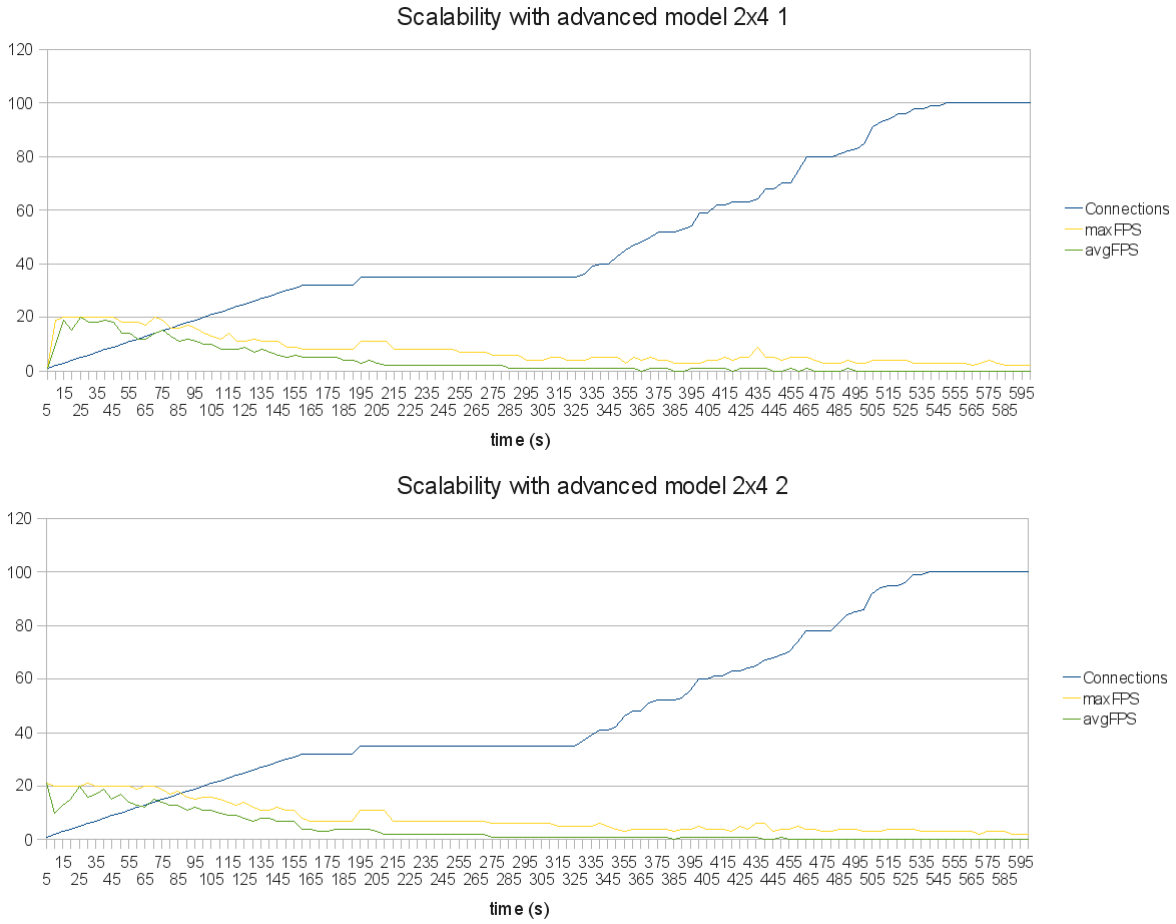


Figure 6.8: Scalability test results using two clients accessing advanced model. Because we had two separate client programs in the tests, we have two different graphs. But as the graphs shows, there is only minor differences between the clients. The test configuration contains two servers and four server processes in each.

Figure 6.7 presents the very same test with the advanced model and the results are significantly worse than with the simple model, but like in the thin-client measurements, the flaws in Simantics and its DEVS implementation causes performance problems when we have more components on the model. The figure shows that the system is not usable after 30 connections to a single server, and after 40 connections the server has difficulties to initialize the incoming connections which shows as variation on the line showing the number of concurrent connections.

Because of the poor performance in the scalability test with the advanced model, we decided to increase the amount of server processes from two to four on each server, hence we have eight server processes serving the incoming 200 connections. Figure 6.8 presents the results of this test. This configuration provides good usability when we have 30 connections to the both physical server and thus 7-8 connections to a single process. However, the system is not capable to access new connections after we have more than 30 connections from the both client programs. This behavior shows that we do have performance related flaws in the Simantics and the DEVS. With four processes running on both servers the system consumes all the server resources and thus the server does not have enough resources to accept new connections.

Because of the flaws in the current version of the Simantics and in the components we have implemented, these measurements are only suggestive, hence the results must be analyzed in a qualitative way.

6.2 Architecture tradeoff analysis

Architecture Tradeoff Analysis Method (ATAM) is a structured technique for understanding the tradeoffs inherent in the software system architectures. This method can be used to evaluate the architecture against the non-functional, quality attribute requirements. ATAM can be done early in the development life cycle, and it is relatively inexpensive and quick method. We use this method to analyze the architecture described in the chapters 4 and 5 against the requirements defined in the chapter 3, and evaluate whether or not the designed architecture is suitable for the system we are developing. The ATAM process is described in the next section, and the actual results are presented after that. (Kazman et al., 2000)

6.2.1 Architecture Tradeoff Analysis Method process

ATAM process consists of nine steps as follows:

1. **Present the ATAM** for the project stakeholders.

2. **Present business drivers** i.e., what business goals are motivating the development effort.

3. **Present architecture** for the stakeholders, focusing on how it addresses the business drivers.

4. **Identify architectural approaches**. Different architectural approaches are presented and discussed.

5. **Generate quality attribute utility tree** from the non-functional requirements (i.e., quality attributes). Each quality attribute will be specified down to the level of scenarios and prioritized.

6. **Analyze architectural approaches**. The architectural approaches that address the high-priority factors identified in the previous step will be elicited and analyzed. During this step, architectural risks, sensitivity points, and tradeoff points are identified.

7. **Brainstorm and prioritize scenarios** with a larger stakeholder community. In this step, each scenario will get importance point.

8. **Analyze architectural approaches**. This step reiterates step 6 with added knowledge of the larger stakeholder community.

9. **Present results** of the information collected in the ATAM for the stakeholders.

The information required for the steps 1-4 is already presented in this work. The quality attribute utility tree for the step 5 is presented in the next section in a table format. The table contains also the updates made in the steps 6 and 8. The results of the information collected in the ATAM will be presented in the next chapter.

(Kazman et al., 2000)

6.2.2 Quality Attribute Utility Tree

Quality attribute	Requirement	Scenario	Priority
Performance	Latency	Latency for the user less than 200ms	H, M
	Bandwidth	Average bandwidth consumption less than 100kbps	H, M
	Bandwidth	Model diagram must be loaded in less than 10 seconds using 3G connection	M, M
	Memory&CPU	The reference server must be able to serve 100 concurrent connections.	M, H
Modifiability	Support for Simantics plugins	Thin-client implementation must not need to be changed if new solvers are added to Simantics with plugins	H, H
	3D support	Support for 3D diagrams implemented in 6 person-months.	M, M
	Secured connection	Secured connection between the client and the server implemented in 4 person-months.	H, M
	Third party servers	Third parties should be able to provide computing power by hosting Simupedia servers.	L, L
	AJAX frontend	AJAX based frontend implemented in 8 person-months.	L, H
Availability	HW failure	Any resource shall not be unavailable more than 5 seconds at a time.	H, M
	Connection queue	If all server resources are in use, the client should be put into a queue.	M, M
Scalability	Amount of users	The system must scale up to 10 000 concurrent users.	H, M
	Amount of models	The system must be able to host at least 1 000 000 models.	M, L
Maintainability	Centralized log system	All logs accessible from a common system in less than 1 minute delay.	M, L
	Customer hosted servers	Standalone installation of the system should be possible to deploy to customers servers in one day without any changes to the code.	H, M
	Dedicated servers	Dedicating a server for a specified user group must be able to be performed in less than 10 minutes.	M, M

Each scenario is prioritized along two dimensions: importance and risk, by using relative ranking High, Medium, Low. In the table above, the first letter in the priority column refer to importance, and the second to risk. Importance refer to importance of achieving the scenario. Risk refer to a risk of not achieving the scenario. Following sections describes the risks, sensitivities and tradeoffs related to the scenarios in more detail. Tradeoff refer to dependencies between attributes, when sensitivity refer to areas of the system that are significantly impacted if the architecture changed.

Performance

Latency for the user must be less than 200ms. This means that when the user for example types a letter into an input box, the letter must be displayed for the user in 200ms. 200ms latency corresponds 5 fps average frame rate, which can be achieved with the designed architecture according to the measurements. For usability, this is a very important scenario, hence this has the highest priority. Risk of not achieving this scenario is medium since the measurements shows variation with latency even though the latency should not be a problem with the required network connections. To minimize the risk we have to optimize our solution to minimize all the latencies caused by the software. Latency is also highly sensitive to any changes in the architecture because any delays in any component will increase the latency. For example, the increase in the amount of concurrent users in a server will probably increase latency.

The average bandwidth consumption from the server to the client must be less than 100kbps. This is important scenario because we must be able to serve enough concurrent users with a limited network bandwidth. Also, the user must be able to use the system with a slow network connection, thus we cannot increase the average bandwidth consumption by improving the server side bandwidth. In order to achieve this scenario, the network traffic must be minimized, and thus nothing must be transferred twice. Even though the measurements shows that this scenario can be achieved, in the future the simulation models are more complicated and contains more interactive components, and thus the risk of not achieving this scenario is medium. The network connection is the main bottle neck in the system, thus everything must be done to optimize the network traffic.

The load time of the model diagram must be less than 10 seconds using 3G connection in order to achieve good usability. This scenario is related to the bandwidth and to the server performance. The client and the server must perform the handshake and the server must be able to initialize the connection and send the first frame in this time. According to the measurements, this will not be a problem, but we have to take in to account the fact that the models will be more complicated in the future. The importance and the risk of not achieving this scenario is evaluated as medium.

To make the solution feasible financially, we must be able to serve 100 concurrent users with a single server. The reference server is described in the scenarios section, and we have calculated that we must be able to serve 100 concurrent users with it. The performance scenarios varies significantly between the models, but these calculations assume that we are using the reference models. This is only medium importance scenario, but the risk of not achieving this scenario is high, since according to the measurement we are not able to achieve this scenario. However, most of the performance issues seem to be caused by flaws in the development version of the Simantics environment. As the measurements shows, the performance is a highly sensitive attribute in the architecture and to optimize the performance of the server we have to consider performance issues in every decision.

Modifiability

The thin-client implementation must be able to handle any kind of server configuration, hence the implementation must be generic and must not need to be customized for each server configuration. The thin-client is the most visible component for the customer, and thus should be kept unchanged as long as possible. In addition, it would be difficult to maintain several different versions of the thin-client. Like said in the previous section, the server configuration is likely to change, and thus this scenario is important to achieve. The risk of not achieving this scenario is high because we cannot know what kind of data the thin-client must be able to display in future when new plugins are developed.

Support for 3D diagrams should be possible to implement in 6 person-months. It has become clear that Simantics simulation environment will have support for 3D models in the future, thus the thin-client should also be capable to display 3D graphics from the server. This is a medium importance scenario. Event though there will be a need for the 3D support, the system will contain mostly 2D models. Although 3D would require major changes to the implementation, this is only a medium risk scenario, since the current scene graph implementation is very much similar to what is widely used in 3D systems (Sowizral, 2000). If the 3D support is decided to be implemented, the AJAX based frontend might not be possible to implement and vice versa, thus this is a tradeoff between the 3D and the AJAX frontend.

Providing secured connection will be important if companies will place sensitive information into the simulation models. In this stage, the secured connection will not be implemented, but the architecture must make it possible. Implementing this kind of secured connection must not take more than 4 person-months. It is very likely that this feature is needed, and thus this is highly important scenario. Risk of not achieving this scenario is medium.

Third parties should be able to provide computing capabilities to be used in the Simupedia public community by hosting Simupedia servers. Because Simupedia is a public web site and is free for non-commercial use, we would like to support third party sponsored servers. For example if Simupedia would be used in a university course, the university could provide servers for Simupedia as a good gesture. This is a low importance scenario because we probably can manage without third party server quite long. The risk of not achieving this scenario is evaluated as low because the architecture supports multiple servers hosted in different locations.

In the future we might want to provide better usability by replacing the Java frontend with AJAX frontend. The architecture must be designed so that the change to the AJAX frontend would not require changes to the whole architecture. This is a low importance scenario, but the risk of not achieving this is high since AJAX systems differ significantly from the Java Applet implementation. Like noticed earlier in this section, we are not able to display 3D graphics in an AJAX frontend, at least not as long as the browser 3D support is not mature enough.

Availability risks

Any resource shall not be unavailable more than 5 seconds at a time. This means that a fault should be detected and the traffic redirected to a working server in 5 seconds. The active connections can be closed, but the user must be able to get a new connection after 5 seconds. This is a high importance scenario because the users must be able to rely on our systems, and paying customers will take the downtimes into account in the service level agreements (SLA). Achieving this scenario requires well designed fault detection, but also resources to keep spare hardware. The risk of not achieving this scenario is evaluated as medium, and this is somewhat sensitive to any changes in the architecture because this feature must be taken into account in every component of the system.

If for some reason all the server resources are in use, the user should be informed and the requested connection should be put into a queue. With this procedure the users are served as soon as possible in a first come first served order. To provide good user experience this is important feature but will not probably be needed very often. This is a medium importance and medium risk scenario.

Scalability

The system must scale up to 10 000 concurrent users. By using the architecture and the components designed in this thesis and increasing the amount of hardware we should be able to serve this amount of users. Achieving this scenario is highly important since

it would cause major costs to redesign the architecture again before we get to 10 000 concurrent users. The risk of not achieving this scenario is medium.

The system must be able to host at least 1 000 000 models. Hosting a single model should not consume resources because we are creating a system that can be used to publish models, hence one of the main purposes of the system is to host models. Achieving this scenario is highly important. The risk of not achieving this scenario is evaluated as medium since a huge model database can cause performance problems in for example connection routing.

Maintainability

All server logs must be accessible from a common log system in less than 1 minute delay. When we have a large number of physical servers and even more server processes running concurrently, it is not feasible to monitor and to find possible problems by browsing log files from their physical location. Therefore the log data must be collected into a centralized system. Collecting log data might cause major network traffic and thus the log system must be designed carefully to scale up. However, this should not be a problem if this is taken into account when the system is designed. This is a medium importance scenario, and the risk of not achieving this scenario is low.

Standalone installation of the system should be possible to deploy to customers server environment in one day without any changes to the code. Many companies do not want to locate sensitive information outside the company, thus they would like to host the Simupedia servers by themselves in case they take Simupedia in use in the company. Therefore the architecture should be so simple that the whole system could be possible to install on a single server. This is a highly important scenario because it is very likely that the first customers do want to host the system by themselves. The risk of not achieving this scenario is evaluated to be medium.

Dedicating a server for a specified user group must be able to be performed in less than 10 minutes. For paying customers we must provide good service, thus it must not take too long to provide the service that the customer has bought. Achieving this scenario is medium importance. The risk of not achieving this scenario is evaluated to be medium.

6.3 Quality of the architecture

In this chapter we evaluated the architecture in quantitative and qualitative way. Quantitative measurements were made against the initial implementations of the components, and thus these results are only suggestive. Qualitative analysis of the architecture gives us better picture about the quality of the architecture because the final system

is not ready yet. We used the Architecture Tradeoff Analysis Method (ATAM) to analyze the architecture and the results show that we have designed a working architecture but we identified potential risks that might occur in the future. By recognizing these risks, we can improve the architecture and the implementation before the system is released.

Chapter 7

Conclusions and future work

This thesis was part of project called Simupedia in which a web based simulation environment is developed. Simupedia brings simulations to general public and also makes it more efficient to apply simulations in businesses. Simupedia concept enables invention of totally new applications for simulation. In addition to the latest collaboration tools and technologies, Simupedia contains a simulation model viewer which was developed in this thesis. This model viewer is a thin-client implementation for Simantics simulation environment, and it makes it possible to easily access simulation models, without need for installing large software packages. Thin-client approach solves also some license issues, because the simulation solvers are located on the server-side and are never redistributed to the client.

In this thesis, we designed the architecture for the thin-client, including the graphics transfer protocol, server load balancing algorithm and the Simantics integration. This work included also the implementation of the thin-client, the server software and the load balancing component. In addition, after the components were implemented, we measured the performance and the scalability of the system. After the measurements, we evaluated the designed architecture with Architecture Tradeoff Analysis Method (ATAM). In the ATAM process the architecture was presented to the other stakeholders of the project and analyzed with them. The measurements performed earlier gave good quantitative data for the evaluation phase, and the data played important role when the scenarios were prioritized and the risks were identified.

Even though this thesis was only a small part of the Simupedia project, we managed to implement a preliminary but well working thin-client solution. According to the ATAM results, the designed architecture is suitable for the system. However, we identified some potential risks that should be taken care of before proceeding with the project. One of these risks is the server performance. The measurements show that the current implementation of the server consumes too much resources, and we are

not able to serve the required amount of concurrent users with one physical server. However, these performance problems are mainly caused by the flaws in Simantics, which are now identified and will be fixed in the future releases. It was a known fact that Simantics is not yet a stable software, thus we were prepared to face problems caused by the flaws and lack of features in Simantics. We did not identify any major performance related problems caused by the architectural decisions made in this work.

In the ATAM it also turned out that the architecture might not work as such with the future features of Simantics. One major improvement to the architecture is support for 3D graphics. Even though our scene graph implementation should work in a 3D environment with small changes, the 3D support brings new performance related problems. The required changes for the 3D support should be implemented as soon as possible, because any architectural changes are difficult to apply when the system is in production. Another unimplemented feature is secured communication with the client and the server. Although the implementation should be quite straightforward, securing network traffic will always cause performance loss, and thus we might have problems achieving the performance requirements with secured communication. At some point we might want to get rid of the Java dependency on the client side, but that will not happen for a while. However, we must prepare the system architecture for that change, and thus the requirements for the AJAX client implementation should be elicited as early as possible.

Even though the implemented components are not final, and they do contain known flaws, the implementation and the architecture are proven to be working. The key technologies for the Simupedia project were designed and developed in this thesis, and thus we are able to bring the simulations to web. With these technologies we have been able to try out the Simupedia concept and the concept has proven to be working and it has commercial potential. Because the whole concept of the web based simulation environment is quite new, we can only guess the use cases for the system, but our targets are high and we do believe that Simupedia will change the world.

Bibliography

Jay April, Marco Better, Fred Glover, James Kelly and Manuel Laguna. Enhancing business process management with simulation optimization. *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 642–649. Winter Simulation Conference, 2006. ISBN 1-4244-0501-7.

Mario Barbacci, Mark H. Klein, Thomas A. Longstaff and Charles B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021 ESC-TR-95-021, Carnegie Mellon University, December 1995.

Thomas Brisco. Dns support for load balancing, 1995. URL `http://www.dns.lu/Pdfs/rfc1794.txt`.

Gerardo Canfora, Giuseppe Di Santo and Eugenio Zimeo. Developing and executing java awt applications on limited devices with tcpte. 2006.

Shuping Cao, John Grundy, John Hosking, Hermann Stoeckle, Ewan Tempero and Nianping Zhu. Generating web-based user interfaces for diagramming tools. *AUIC '05: Proceedings of the Sixth Australasian conference on User interface*, pages 63–72, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920682-22-8.

Vidyanand Choudhary. Software as a service: Implications for investment in software development. *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 209a–209a, Jan. 2007. doi: 10.1109/HICSS.2007.493.

George Coulouris, Jean Dollimore and Tim Kindberg. *Distributed Systems - concepts and design*. Addison Wesley, Harlow, fourth edition edition, 2005. ISBN 0-321-26354-5.

S.G. Eick, M.A. Eick, J. Fugitt, B. Horst, M. Khailo and R.A. Lankenau. Thin client visualization. *Visual Analytics Science and Technology, 2007. VAST 2007. IEEE Symposium on*, pages 51–58, 30 2007-Nov. 1 2007. doi: 10.1109/VAST.2007.4388996.

James O. Henriksen, André Hanisch, Stefan Osterburg, Peter Lorenz and Thomas J. Schriber. Web-based simulation 1: web based simulation center: professional support for simulation projects. *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 807–815. Winter Simulation Conference, 2002. ISBN 0-7803-7615-3.

Patricio Inostroza and Jacques Lemordant. Application-level graphic streaming channel. *LANC '03: Proceedings of the 2003 IFIP/ACM Latin America conference on Towards a Latin American agenda for network research*, pages 99–105, New York, NY, USA, 2003. ACM. ISBN 1-58113-789-3. doi: http://doi.acm.org/10.1145/1035662.1035671.

Donald W. Johnson and T. J. Jankun-Kelly. A scalability study of web-native information visualization. *GI '08: Proceedings of graphics interface 2008*, pages 163–168, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0.

R. Kazman, M. Klein and Paul Clements. Atam: Method for architecture evaluation, 2000.

Markus Ketterl, Robert Mertens and Oliver Vornberger. Vector graphics for web lectures: Comparing adobe flash 9 and svg. *Ninth IEEE International Symposium on Multimedia 2007 - Workshops*, pages 389–395, 2007. ISBN 9780-7695-3084-0.

T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *Software Engineering, IEEE Transactions on*, 17(7):725–730, Jul 1991. ISSN 0098-5589. doi: 10.1109/32.83908.

Tuukka Lehtonen. Ontology-based diagram methods in process modelling and simulation. Master's thesis, Helsinki University Of Technology, 2007.

Macromedia. Using flash remoting mx, 2002. URL `http://download.macromedia.com/pub/flash/flashremoting/using_flash_remoting.zip`. Accessed 6.9.2009.

Farshad Moradi, Rassul Ayani and Imran Mahmood. An agent-based environment for simulation model composition. *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 175–184, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3159-5. doi: http://dx.doi.org/10.1109/PADS.2008.18.

Juan Andrés Negreira, Javier Pereira, Santiago Pérez and Pablo Belzarena. End-to-end measurements over gprs-edge networks. *LANC '07: Proceedings of the 4th international IFIP/ACM Latin American conference on Networking*, pages

121–131, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-907-4. doi: http://doi.acm.org/10.1145/1384117.1384134.

W. Qiao, M. McLennan, R. Kennell, D.S. Ebert and G. Klimeck. Hub-based simulation and graphics hardware accelerated visualization for nanotechnology applications. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1061–1068, Sept.-Oct. 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.150.

G. Reitmayr and D. Schmalstieg. Flexible parametrization of scene graphs. *Virtual Reality, 2005. Proceedings. VR 2005. IEEE*, pages 51–58, March 2005. doi: 10.1109/VR.2005.1492753.

simantics.org. Simantics documentation, 2009. URL `https://www.simantics.org/wiki/index.php/Main_Page`. Accessed 20.7.2009.

H. Sowizral. Scene graphs in the new millennium. *Computer Graphics and Applications, IEEE*, 20(1):56–57, Jan/Feb 2000. ISSN 0272-1716. doi: 10.1109/38.814563.

Sun Microsystems. Applets, 2009. URL `http://java.sun.com/applets/`. Accessed 6.9.2009.

Sun Microsystems. Graphics2d, 2007. URL `http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Graphics2D.html`. Accessed 20.7.2009.

Sun Microsystems. Understanding network class loaders, 2004. URL `http://java.sun.com/developer/technicalArticles/Networking/classloaders/`. Accessed 20.7.2009.

Anucha Tungkasthan, Pittaya Poompuang and Wichian Premchaiswadi. Smile visualization with flash technologies. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD 08. Ninth ACIS International Conference on*, pages 551–556, 2008. ISBN 978-0-7695-3263-9.

M. Turner, D. Budgen and P. Brereton. Turning software into a service. *Computer*, 36 (10):38–44, Oct. 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1236470.

B. Vankeirsbilck, P. Simoens, J. De Wachter, L. Deboosere, F. De Turck, B. Dhoedt and P. Demeester. Bandwidth optimization for mobile thin client computing through graphical update caching. *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian*, pages 385–390, Dec. 2008. doi: 10.1109/ATNAC.2008.4783355.

S.Jae Yang, Jason Nieh, Matt Selsky and Nikhil Tiwari. The performance of remote display mechanisms for thin-client computing. *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

Chunkyun Youn. Performance improvement of cluster system by server status information. *Computer and Information Science, 2005. Fourth Annual ACIS International Conference on*, pages 282–287, 2005. doi: 10.1109/ICIS.2005.102.

Chunkyun Youn and Ilyoung Chung. An efficient load balancing algorithm for cluster system. *Network and Parallel Computing*, 2005. doi: 10.1007/11577188_23.

Bernard P. Zeigler, Herbert Praehofer and Tag G. Kim. *Theory of Modeling and Simulation, Second Edition*. Academic Press, second edition, January 2000. ISBN 0127784551.

J.S. Zepeda and S.V. Chapa. From desktop applications towards ajax web applications. *Electrical and Electronics Engineering, 2007. ICEEE 2007. 4th International Conference on*, pages 193–196, Sept. 2007. doi: 10.1109/ICEEE.2007.4345005.