NTNU
Innovation and Creativity

Supporting SAM: Infrastructure Development for Scalability Assessment of J2EE Systems

Geir Bostad

Master of Science in Informatics
Submission date: October 2006
Supervisor: Peter Hughes, IDI
Co-supervisor: Gunnar Brataas, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The subject of this cand.scient thesis is the exploration of scalability for large enterprise systems. The Scalability Assessment Method (SAM) is used to analyse scalability properties of an Internet banking application built on J2EE architecture.

The report first explains the underlying concepts of SAM. A practical case study is then presented which walks through the stages of applying the method. The focus is to discover and where possible to supply the infrastructure necessary to support SAM.

The practical results include a script toolbox to automate the measurement process and some investigation of key scalability issues. A further contribution is the detailed guidance contained in the report itself on how to apply the method.

Finally conclusions are drawn with respect to the feasibility of SAM in the context of the case study, and more broadly for similar applications.

Preface

This paper is a Cand.scient thesis conducted at IDI, NTNU during October 2003 to October 2006. The thesis is credited the work of two semesters. Cand.scient is a former degree in the Norwegian university system, and it is now replaced by the Master degree.

I would like to express my gratitude towards my advisor Peter Hughes (IDI, NTNU) for his excellent and professional guidance throughout the project. We had numerous of interesting discussions on the scalability subject, and his experience in the performance evaluation field has been invaluable. I would also like to thank my co-advisor Gunnar Brataas (IDI, NTNU) for valuable inputs and long discussions about performance modelling.

I would like to thank EDB Business Partner A/S for giving us the opportunity to use their framework and example application for this case study.

The Clustis staff was very helpful in supporting our measurements needs. Thanks go to Jörg Cassens, Jan Christian Meyer, Zoran Constantinescu and Robin Holtet.

Thanks also go to PhD-student Jakob Sverre Løvstad who was also very helpful in answering my questions.

Finally I would like to thank my fellow students over the years: John Arne M. Fagerli, Odd Christian Landmark, Erik Rød, Erlend Mongstad, Jørgen Ruud, Olav Tveiten and Anders Holmefjord.

Trondheim, 29nd October 2006

Contents

1 Introduction **12**
- 1.1 Project description . . . 12
- 1.2 Modifications to the project description . . . 13
- 1.3 Structure of the thesis . . . 13
- 1.4 Research methods . . . 13
- 1.5 Criteria . . . 14
- 1.6 Research context . . . 14
- 1.7 Project evolution . . . 15

I Foundation **16**

2 Application domain **17**
- 2.1 BankApp . . . 17
- 2.2 Transigo . . . 18
- 2.3 System architecture . . . 18
- 2.4 Using BankApp as case study . . . 18
- 2.5 Java Virtual Machine . . . 19
- 2.6 Java garbage collectors . . . 20
- 2.7 Garbage collector logging . . . 21
- 2.8 Web application architecture . . . 21
- 2.9 Server topology . . . 22
- 2.10 Typical production environment . . . 23

3 Scalability **24**
- 3.1 Introduction . . . 24
- 3.2 Motivation for scalability analysis . . . 24
- 3.3 Scalability vs. performance evaluation . . . 25
- 3.4 General scalability definition . . . 25
- 3.5 Work and load . . . 26
- 3.6 Size vs capacity . . . 26
- 3.7 Scalability function definition . . . 27
- 3.8 Dimensions of scalability . . . 27
- 3.9 Scaling function . . . 27
- 3.10 Hierarchic scaling . . . 28
- 3.11 Non-linear sources . . . 29
- 3.12 Non-linear effects . . . 29
- 3.13 Operating point . . . 30
- 3.14 Scaling consistency . . . 30
- 3.15 Baseline and upgraded system . . . 31
- 3.16 Operational analysis . . . 31

4 Scalability Assessment Method **33**

4.1 Modelling overview . . . 33
4.2 SAM procedure . . . 34
4.3 SAM main steps . . . 35
4.4 Modelling work and load . . . 36
4.5 Static model: SP . . . 36
4.6 Dynamic model . . . 39
4.7 Baseline model: static and dynamic model combined . . . 39

II Case Study **41**

5 Applying SAM **42**
5.1 Context basis . . . 42
5.2 The phases of SAM . . . 42

6 Platform and software **44**
6.1 Baseline configuration . . . 44
6.2 Clustis2 . . . 45
6.3 Old nodes on Clustis2 . . . 45
6.4 Available nodes . . . 46
6.5 Software . . . 46

7 Scaling objectives **49**
7.1 Operational context . . . 49
7.2 Software and hardware scope . . . 49
7.3 Scaling . . . 50
7.4 Scale invariants . . . 50

8 Workload specification **52**
8.1 Top level operations . . . 52
8.2 Transaction . . . 52
8.3 Work-mix . . . 53
8.4 Concurrent user sessions and think time . . . 53
8.5 Load . . . 54
8.6 Implementing the workload with Grinder . . . 54

9 Measurement toolbox **58**
9.1 Why build a toolbox script library . . . 58
9.2 Measurement tasks . . . 58
9.3 Cluster resource limitations . . . 59
9.4 Overview . . . 59
9.5 Structure of the toolbox: . . . 60
9.6 List of toolbox script files . . . 60
9.7 Experiment overview . . . 61
9.8 How to run a new experiment . . . 63
9.9 Toolbox in action . . . 63
9.10 Toolbox scope and extensibility . . . 65

10 Pilot measurements **67**
10.1 Test harness overview . . . 67
10.2 Performance indices . . . 68
10.3 Pilot measurements . . . 68

10.4 Pilot measurements results . . . 69
10.5 Experiment variation . . . 72
10.6 Steady state: arrival rate . . . 72
10.7 Steady state: long runs . . . 73
10.8 Steady state: garbage collector investigation . . . 75
10.9 Experiment problems . . . 76
10.10 Comparison of old and new interval design . . . 77
10.11 Choosing the operating point . . . 78

11 Investigation of non-linearities **80**
11.1 Load dependence vs non-linearity . . . 80
11.2 Static model and load dependent variables . . . 80
11.3 Groups of non-linear effects . . . 81
11.4 Possible non-linear sources . . . 81
11.5 Service demands on upgraded nodes . . . 81
11.6 Garbage collector service demands . . . 83
11.7 Garbage collector non-linear effects . . . 84
11.8 Application server focus . . . 84
11.9 Paging . . . 86
11.10 Connection pooling . . . 87

12 SP model construction **89**
12.1 Rationale for simplicity . . . 89
12.2 The process . . . 89
12.3 SP model construction . . . 89
12.4 The BankApp SP model . . . 90
12.5 Operation types . . . 91
12.6 Complexity function investigation . . . 92

13 SP model parameters **93**
13.1 Calibration vs. parameter measurements . . . 93
13.2 Top-level operation parameters . . . 94
13.3 Garbage collection and heap size . . . 96
13.4 Increased database . . . 98
13.5 Implementing the static model . . . 99

14 Dynamic model **100**
14.1 Analytic approach . . . 100
14.2 Open vs. closed queuing networks . . . 100
14.3 Closed single class queuing network . . . 101
14.4 Open multiclass queuing network . . . 102
14.5 Alternative model: simulation . . . 104
14.6 Baseline validation . . . 104

15 Projecting the upgraded model **106**
15.1 Scale factor investigation . . . 106
15.2 CPU comparison . . . 106
15.3 Possible scaling factors . . . 106
15.4 Measuring the scale factor . . . 107
15.5 Determine the scale factor . . . 108
15.6 Modification analysis . . . 110

15.7 Validation of projections . . . 111
15.8 The gain of the upgraded system . . . 112

III Feasibility **113**

16 Experiences in applying SAM **114**
16.1 Phase 1: Establishing the baseline . . . 114
16.2 Phase 2: Modelling the baseline . . . 114
16.3 Phase 3: Scalability of the upgraded system . . . 117
16.4 Further work . . . 117

17 Conclusion **121**
17.1 Summary . . . 121
17.2 Problems encountered . . . 121
17.3 General feasibility of SAM . . . 121
17.4 Suggestion for SAM improvement . . . 122
17.5 Main contribution . . . 122
17.6 Goal achievement . . . 124

IV Appendix **127**

A Parameter calculation **128**

B Measurement results **131**
B.1 Connection pooling results . . . 131

C Problems encountered **132**
C.1 Two separate clusters . . . 132
C.2 Problems identified by measurements . . . 132
C.3 Problems with repeating FL measurements . . . 133
C.4 Workmix script user limitation . . . 133
C.5 Error in workmix script . . . 134
C.6 Tomcat threads . . . 134
C.7 IOstat disk utilisation . . . 135
C.8 Loading images with Grinder . . . 135

D Work on measuring integrated configuration **136**
D.1 Integrated configuration . . . 136
D.2 Service demands . . . 136
D.3 CPU usage . . . 137
D.4 Session service demands . . . 138

E Work on accessing EJBs directly with Grinder **139**
E.1 Progress . . . 142

F Capturing network parameters **143**

G Work on transforming the SP model **145**
G.1 Simple model . . . 145
G.2 Fagerlie-Landmark model . . . 145

G.3 Intuitivity 146
G.4 MVC in SP 146

H Toolbox load generation **149**
H.1 Grinder workload script 149
H.2 Database generation scripts 151

I Toolbox scripts **153**
I.1 analyse-sar-log.pl 153
I.2 analyse-grinder-logs.pl 154
I.3 analyse-gc-log.pl 157

J Simulating the dynamic model **160**
J.1 Open multiclass simulation 160

List of Tables

2.1 Web application tiers and roles 23

4.1 Variable types in a static model 38
4.2 Complexity functions in a static model 38

8.1 Think times of a typical Bankapp user 54

10.1 Comparing service demands for the two interval designs 78

11.1 Garbage collector service demands for baseline and upgraded system 84
11.2 Measuring 100 idle connections 88

13.1 Parameter measurement results 95
13.2 Comparing garbage collector on baseline and upgraded system 97

14.1 Session response time reported by Grinder 105
14.2 Baseline validation results 105

15.1 CPU specification comparison 106
15.2 Upgraded nodes: Service demands for single user measurements. Runlength is 10 000. 108
15.3 Upgraded nodes: Service demands for single user measurements. Runlength is 30 000. 108
15.4 Session resource demands for each server 108
15.5 New scalefactor 109
15.6 Upgrade validation results 112

D.1 Service demands for integrated web and application server 137

List of Figures

1.1 Research context timeline 14

2.1 BankApp screenshot 17
2.2 Transigo mapped to system architecture 18
2.3 Four garbage collectors provided by Sun JVM 20
2.4 Typical web application configuration 23

3.1 Figure 3.1The scaling function 28
3.2 Three hierarchical levels of scaling options 29

4.1 Modelling overview 33
4.2 SAM procedure. Figure by Peter Hughes, 2006 35
4.3 SP 37
4.4 SP complexity matrices 37
4.5 Baseline model comprises of static and dynamic model 40

6.1 Baseline configuration 44
6.2 Modified baseline for increasing load on application server 45

8.1 Workload implemented as Grinder threads on one node 55

9.1 Experiment.sh 62

10.1 CPU utilisation for baseline and upgrade 70
10.2 Baseline Web server: Comparing various measurement utlilisation results 70
10.3 Garbage collector utilisation for baseline and upgrade 71
10.4 Request response times for baseline and upgrade 71
10.5 Upgraded system: Session response time (left) and Session throughput (right) 72
10.6 CPU utilisation on web server. Experiment 1 (left) and experiment 2 (right) 72
10.7 Customers arriving pr. 60 seconds (left), pr. 20 seconds(middle) and pr. 1 second in shorter interval (left) 73
10.8 Web node CPU utilisation. Figure to the right shows the data in a shorter interval. . . 73
10.9 Web server stats CPU, Network, Memory 74
10.10 App server stats (a) CPU (b) Network (c) Memory 74
10.11 DB server stats (a) CPU (b) Network (c) Memory 75
10.12 Garbage collector activity Web node (left), App node (right) 76
10.13 Closer look on garbage collector activity 76

11.1 Upgraded system: Service demand per session. Heap=700MB 82
11.2 Upgraded system: Service demand per session. Heap=175MB 82
11.3 Upgraded web server: Garbage collector service demand 83
11.4 Upgraded app server: Garbage collector service demand 84
11.5 Modified baseline: Application server focus. 85
11.6 Baseline application server: Servide demand pr. session 85
11.7 Baseline application server: Garbage collector service demands 86

11.8 Upgraded web server(left) and app server(right): Paging statistics for heap=900MB . 87

12.1 SP model 91
12.2 Mapping of Java methods to SP components 92

13.1 Upgraded web server: Garbage collector service demands when heap size increases . 97
13.2 Database response time for increased data-load 98
13.3 Comparing server CPU utilisations with increased user base 99

14.1 Dynamic model spreadsheet 101
14.2 Dynamic model results 102
14.3 Dynamic model results 102
14.4 Dynamic model spreadsheet 103
14.5 Dynamic model results 103
14.6 Dynamic model results 104
14.7 Results from BankApp simulation 104

15.1 Comparing service demands for baseline and upgraded 109
15.2 Dynamic model of upgraded system 110
15.3 Dynamic model of upgraded system: Utilisation results 111
15.4 Dynamic model of upgraded system: Response time results 111

A.1 Excel spreadsheet for calculating service demands 129

C.1 Comparing FL devolved work to measured service demands 133

D.1 CPU usage for integrated configuration 137
D.2 Service demands on intergrated web and appserver 138

F.1 Calculating network usage using a spreadsheet 144

G.1 SP - simple model 145
G.2 Fagerlie-Landmark SP model 146
G.3 Model View Controller (SUN blueprints) 147
G.4 SP - collapsed view of a transformed model 148
G.5 SP - complete view of a transformed model 148

1 Introduction

Scalability of IT systems is described with focus on J2EE architecture. Analysing scalability will be performed in the context of a web application case-study.

EDB Business Partner has supplied an application called BankApp. This is an Internet banking application allowing users to perform basic operations such as “log on”, “view payment history” and “make new payment”. BankApp is an example application developed by EDB to demonstrate how to use their own Transigo framework. Transigo is a library built on J2EE.

The Scalability Assessment Method (SAM) [1] is developed by Peter H. Hughes at NTNU. The method helps answering questions like: Will an upgraded system support increased load in an economical fashion? The method is intended to aid software engineers to consider scalability when developing systems. It is a tool to predict and hopefully avoid dis-economies of scale when systems are upgraded as load increases over time. From a business point of view, the management staff want value from their new hardware investments. Bad architectural design may lead to wasted hardware resources because of scalability issues.

We show how to model the the BankApp transaction processing system and make predictions of how an up-scaled version of this system will perform with increased load. Finally, the model predictions are validated by measurements.

1.1 Project description

The two first paragraphs sets the context for the project, and the last paragraph describes the project task as originally conceived:

> Scalability of information processing systems describes the relationship between the delivery of services and the amount of physical resources needed to support them. Both services and resources may be characterised in the dimensions of processing, communication and storage. The scalability of a system as a whole depends upon the scalability of its software and hardware components in a complex way. Scalability within the three dimensions and within different layers of software can all interact. Architectural, algorithmic and implementation properties all play a part.
>
> The methodology being developed depends on separating out the various causes of non-linearity and then using performance modelling to analyse the effects of interactions between them. The potential value of this lies in the contribution it can make to the process of developing and integrating large-scale distributed systems with predictable performance and scaling properties.
>
> The task is to investigate and classify the possible causes of non-linear behaviour in large information processing systems, and to use this classification to test the overall (SAM) methodology. *This paragraph was changed, see the next section.*

1.2 Modifications to the project description

However, early experience with the experimental aspects of SAM showed that the lack of a supporting infrastructure was a serious obstacle to progress. The project objective was therefore changed and given a more practical slant. This will be explained in more detail in Section 1.7. The last paragraph of the project text was modified to this final version:

> "The task is to develop a supporting infrastructure for SAM in the practical context of a J2EE-based banking application, and hence to investigate the feasibility of the overall methodology".

1.3 Structure of the thesis

Part I: Foundation

In chapters 2 - 4, a summary of the foundation and theory for the project is given. Topics are briefly outlined and background material is discussed and referenced to. In Chapter 4 we present a general briefing of the Scalability Assessment Method method.

Part II: Case study

Chapter 5 presents an index to how we apply SAM in the case study. The case study is presented in chapters 6 - 15. The modelling, analysis and actual measurements are described in detail.

Part III: Feasibility

Chapter 16 presents a discussion of the feasibility of SAM in context of the case study. Problems, challenges and solutions are discussed. What is accomplished, and what parts are useful for further work on this topic. In Chapter 17 the thesis is summarised, and general conclusions for SAM are drawn from the case study experiences.

1.4 Research methods

The basis for our research is the Scalability Analysis Method, see Chapter 4. The method describes how to perform a scalability analysis in a formal way. SAM is a relatively new method and hence the need to be tested on real-world applications to analyse the feasibility of the method in various domains. Ultimately, this method is intended to address challenges that the industry often meets, such as aiding developers in analysing performance and scalability as early as in the development phase.

The goal is to assess and possibly improve SAM by:

- **Apply** the method to an application domain.
- **Evaluate** the techniques and procedures used in SAM.
- **Improve** the method by identifying problems and challenges and propose changes.
- **Document** enough details on how to actually get the results. Describe what approximations are used, and why are they used. The aim is to support and guide subsequent researchers and developers in applying the method.

1.5 Criteria

These are the criteria making the thesis successful:

1. Describe in a detailed fashion how to apply SAM to a system, and capture the considerations behind the choices.
2. Formalise the method and provide a toolbox. This enables peer students to focus on bringing the Scalability Assessment Method one step further, rather than doing much of the same work all over again.
3. Apply SAM to investigate key scalability issues for this case study. The focus is on the processing dimension, but storage and connectivity dimension must also be investigated.

1.6 Research context

The cand.scient grade counts for two semesters of work and is much more relaxed in terms of delivery date than a master's thesis. Since this project spanned over several years, a few similar diploma projects were completed in the meantime. The diploma projects count for one semester, and have stricter limits for the deadline.

This project is based on the work of Fagerlie and Landmark. Some work from my project was used in the three other diploma projects, and valuable knowledge and experience was exchanged between us. Interaction also took place in the SPlight and SAMe tool development, so it is natural discuss the context SAM was applied and improved in.

A figure of the timeline can be seen in Figure 1.1.

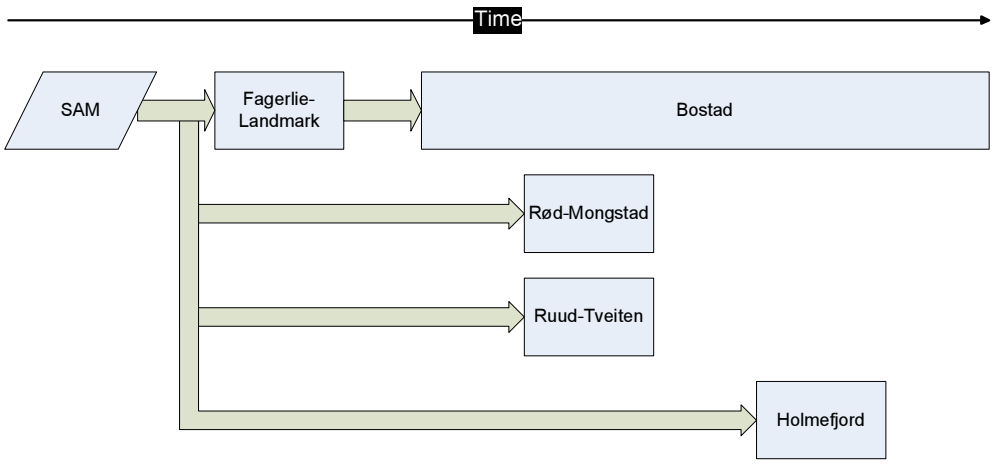


Figure 1.1: Research context timeline

The basis of all five projects is the Scalability Analysis Method. The method has been refined and developed further in the context of the projects:

- **Fagerlie-Landmark**[9]**:** The first students to apply a coarse version of SAM to a case study was Fagerlie and Landmark . Their study is a model-driven approach, where they built a detailed

and complex SP model of BankApp, and in a sense pioneered the work of how to apply SAM to transaction processing systems. The Fagerlie-Landmark master's thesis was used as case study in the article "Exploring the Scalability of an Enterprise Architecture", see[8]. The article summarises the status of the Scalability Assessment Method at that time.

- **Rød-Mongstad**[10]**:** In their diploma thesis, Erik Rød and Erlend Mongstad used SAM to perform a model-driven study of another application from EDB Bank og Finans, called KBM (KredittprosessBusinessMarket).
- **Ruud-Tveiten**[11]**:** At the same time, Jørgen Ruud and Olav Gisle Tveiten used the SAM in their diploma thesis to study a read-intensive web system, loosely modeled after a news system from TV2 Interactive.
- **Holmefjord**[12]**:** Finally, Anders Holmefjord implemented a prototype of SAMe, the SAM Engine, a tool for exploring scalability. See [2] for an outline of the SAM Engine prototype. The goal of this tool is to use performance data from a static model (modelled in SPlight), and run dynamic models to predict performance of upgraded systems. SAM Engine uses simulation as dynamic models at the moment. The simulation of BankApp in Appendix J was used as a basis for the implementation of model simulation in SAMe.

Jakob Sverre Løvstad is in the process of writing his PhD, showing how to incorporate SP as an addon to the UML language. The UML deployment view has limited support for expressing software mapped to hardware, which limits the level of detail in the model. Since SP is designed and well-suited for that purpose, the goal of his work is to incorporate SP to the UML modelling language.

To be able to effectively model in SP, a tool had to be made. In the autumn 2005, an SP modelling tool called *SPlight* was built by five students in a "Customer Driven Project".[1] In the summer 2006 two of the students, Erik Drolshammer and Per Ottar Ribe Pahr, were hired to fix the bugs of SPlight and finish off the project.

I was involved through the whole process of SPlight since I had experience with SP modelling. The involvement included requirement discussions, suggestions and testing. Since I also had guided Anders Holmefjord on his SAMe project and had SAM experience from my own project, I could contribute in how to integrate the tools. Ultimately these tools will comprise a package for aiding a developer in using SAM, and automating several steps of the SAM process.

1.7 Project evolution

All projects evolve and may change from the initial assumptions, and this project is no exception.

A prior requirement was discovered that had to be fulfilled before work could be done on the original text. Infrastructure was needed to be able to perform controlled and repeatable experiments. Even more, infrastructure was needed especially in the configuration and debugging phase of the system. Even configuring a simple system as BankApp required a lot of debugging.

The experience from that process, combined with the fact that neither Fagerlie-Landmark, and Rød-Mongstad did get to finish off their measurements because of technical work, indicated a strong need for infrastructure support. Both support in terms of scripts, but also in terms of a guide of how to perform an analysis based on SAM. A result of this process was that all the nitty-gritty steps in getting the system up and running was automated. Another result is that the report is also a guide on how to perform scalability assessment.

[1] This is a subject at NTNU (TDT4290), where students are assigned projects supplied by external or internal customers.

Part I

Foundation

2 Application domain

This chapter introduces the application domain by describing the technologies referred to and how they relate to each other. First a brief introduction is given of the software that constitutes the case study, explaining how the software relates to the rest of the system. Then general concepts of web application architecture are described. Finally the possible ways to map architecture components to computer nodes are outlined.

2.1 BankApp

EDB Bank & Finans provided the example application for the case study. It is an internet banking application that provides services to a typical bank customer. A customer can perform two main tasks: view payment info and add new payments. These tasks consist of a workflow of several steps. To add a payment for instance, the customer first logs in and then requests the page "new payment" to fill in payment data. He then clicks the "next" button to get a page where he must confirm the payment. Finally, the customer gets a payment receipt page. Figure 2.1 shows a screenshot of the "new payment" page.

The possible interactions with the system is described in Chapter 8. Also see chapter 2 in [9] for more details about BankApp and Transigo. A detailed description of the workflow is given in chapter 5 in [9].

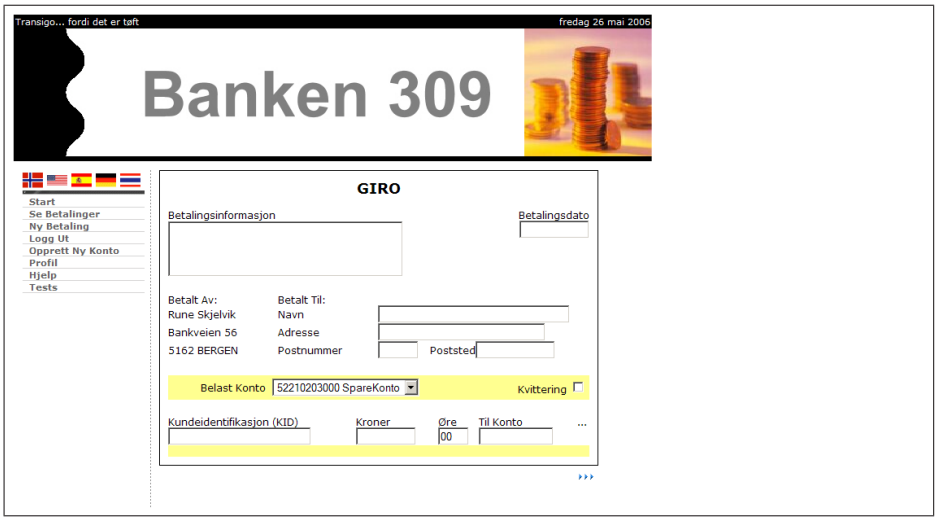


Figure 2.1: BankApp screenshot

2.2 Transigo

The BankApp example application was built on the Transigo framework[1]. Transigo is a framework for developing transaction oriented applications. It is library of routines and classes. The transigo architecture is specified in the Transigo architecture documents [13].

J2EE gives the developer extensive freedom in choosing solutions when implementing an application. Transigo is intended to limit this freedom to a certain degree, forcing the developer to follow well-proven patterns to meet the quality requirements. The Transigo documentation provides certain guidelines that a developer must follow.

2.3 System architecture

A general and simplified view of the system is presented in Figure 2.2 which describes the hierarchy of the system architecture. The Transigo layer incorporates various technologies(or interfaces to them), as pure Java classes, servlets, beans and Java Server Pages.

All Java software runs in the Java Virtual Machine. The bottom layers contains the operating system, which is responsible for creating processes or threads for the JVM to run in.

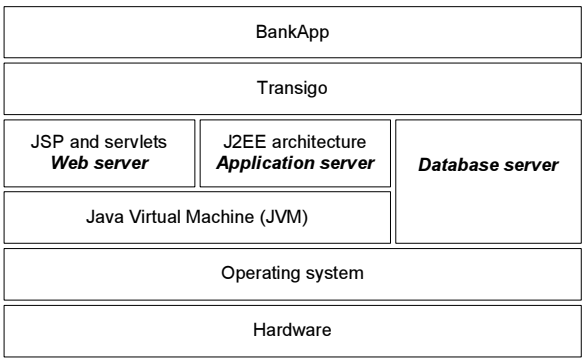


Figure 2.2: Transigo mapped to system architecture

2.4 Using BankApp as case study

As mentioned in previous sections, the BankApp example application is a toy application, where there is little actual business logic behind the requests. The application is merely a shell to give developers an example of how to use the framework to create their own applications.

Requests such as creating new users or viewing customer payments are basically just forwarded as SQL messages from the application server to the database server. After the database server has processed the request, the HTML response is prepared and sent back to the user. Therefore, the requests on the web tier are most demanding, since this is where the JSP pages are generated.

[1]The Transigo framework is not used anymore by EDB Bank & Finans A/S

This means that we are able to stress the architecture rather than the business layer. A higher load (in terms of user sessions) is required to saturate the running system, as opposed to a case with more complex business logic. See Section 8.4 for an explanation of user sessions and concurrency.

Compare these two scenarios where the systems are equally utilised in terms of CPU utilisation[2]:

1. 1000 concurrent user sessions on the system.
2. 100 concurrent user sessions, where each session requires 10 times more CPU resources than case 1. In case 2, more complex business logic is involved.

The goal is to some degree balance the utilisation of the computer resources. We want to stress the system in both in the memory and communication dimension, as well as the CPU usage in the processing dimension.

Assume that the CPU resource usage is measured to be 60% in both scenarios. To investigate scalability of an architecture, it may be more interesting to deal with case 1. The load is 10 times higher, the memory usage may be increased up by the same factor, and a larger number of database connections may be needed.

2.5 Java Virtual Machine

Since BankApp is a Java application, it imposes some challenges with respect to measuring the resource usage for software components. When building a model of a system we need to parameterise the model with resource usage data. Depending on the granularity of the model, we may need to measure the system on a component level rather than system level. Component level measurements are typically performed by profiling the application classes, but such profiling influences the system. There are also issues of how well the profiler tools supports to obtain the resource usage of collections of classes.

A compiled application, can relatively easily be monitored in order to obtain the resource usage of the various software components. There exists a Linux kernel driver which can profile all running code at low overhead[3].

Java applications run in their own virtual machine, the Java Virtual Machine (JVM). This means that a kernel driver profiler would only see one or more processes implementing the virtual machine, not the actual running application. To obtain the resource usage for Java classes, one have to rely on Java instrumenting tools(profilers) to get the performance data.

From Java 2 Release 1.3, the Sun Hotspot JVM implementation uses system threads to implement Java threads, instead of “green threads”. Green threads emulated multithreaded environments, but could only run on one CPU. Different implementations of the JVM might implement Java threads differently.

The new Hotspot JVM is designed for intense server applications. The JVM runs in a mixed runtime mode, meaning that Hotspot dynamically compiles Java byte code into native machine code when a number of criteria have been met, such as the number of invocations of a method in the interpreter.

Since Linux system threads are implemented as cloned processes, Java threads show up in the process table as processes when running the ps(1) command. Native threads create the appearance of

[2]This simple example assumes that the application is CPU-bound in both cases, and that there are no measurable non-linear effects. If there were any non-linearities, the utilisation would probably be higher in case 1 because of the higher data-load.

[3]OProfile is a sourceforge project that can profile all running code: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications. The project can be found at `http://oprofile.sourceforge.net/`

multiple Java processes, when in fact it is one process. A clue that these are all threads of the same process is that they use the same memory, and the process table show that the memory sizes are equal for all entries. Most implementations of the JVM run as a single process.

2.6 Java garbage collectors

Another challenge in measuring Java application performance is garbage collection. A Java garbage collector runs independently of the application, but is a part of the JVM. The resource usage of the garbage collector must be accounted for when measuring the application performance. See [24] for information about SUN Java's garbage collection.

The garbage collector runs as a thread which automatically reclaims unused memory that will never again be accessed. These are the four available garbage collectors from Sun's Java implementation, see[17]:

- Copying collector (available in all J2SEs)
- Mark-compact collector (available in all J2SEs)
- Parallel collector (JSE1.4.1 and higher)
- Concurrent mark-sweep collector (JSE1.4.1 and higher)

The key problem with a garbage collector is that the running program must be stopped while objects are garbage collected. This is a "stop-the-world" pause, and can affect performance in terms of response time. Any garbage collector pause will simply stop the application, which really affects applications that has close to real-time requirements.

The telco industry needs responsive applications with a high throughput. The problem arose when applications needed several gigabytes of memory, and the GC pause made the application very non-responsive. The standard garbage collectors proved insufficient, so two new garbage collectors were implemented in version JSE1.4.1 of Java. The new garbage collectors adressed the realtime needs in the telco business. The industry also wanted to utilise servers with several CPUs, so the new garbage collectors were implemented multithreaded.

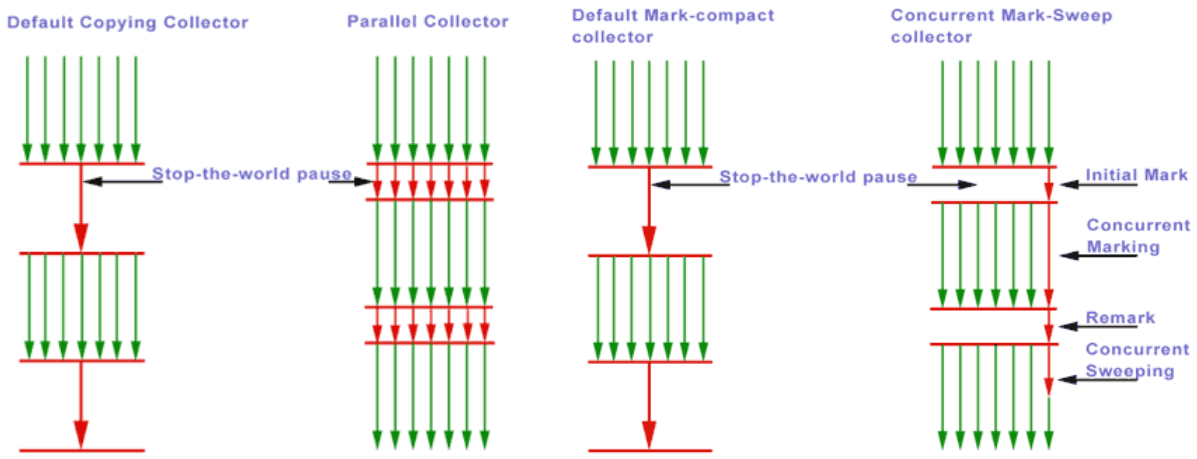


Figure 2.3: Four garbage collectors provided by Sun JVM

Figure 2.3 shows the different garbage collectors provided by the Sun JVM. The figure is taken from [24]. The green arrows represent a multithreaded application running on a computer with multiple

CPUs. The red arrows represent garbage collection threads, and the length indicate the duration of the garbage collection pauses. The default garbage collector is the "copying collector". In production environments, the new parallel collector and the concurrent mark-sweep-collector are enabled to run together, to achieve high throughput and low latency.

In short, the new collectors enable the garbage collection process to monitor in parallel with the application, called concurrent marking. They also have a multithreaded stop-the world pause which makes the pauses shorter. The deployer of an application can choose what garbage collector to use, and tweak it for a given problem domain. This is however a time-consuming task, and Sun tries to adress this issue in JSE1.5, providing a more self-configuring garbage collector.

In this project we choose the standard garbage collector for simplicity. Our focus is how to quantify the resource usage, not tweak the system for best performance.

2.7 Garbage collector logging

This section describes how we obtain resource usage from the garbage collector.

The garbage collector can be configured by passing parameters to the JVM. Some JVM switches produces a log of garbage collector activity. This is an example of garbage collector output. Such a log is used to extract the CPU resource usage of the garbage collector:

Listing 2.1: Output from gc-tomcat.log

```
0.000: [GC 0.000: [DefNew: 63744K->2863K(71680K), 0.0397200 secs] 63744K->2863K(708864K), 0.0397650 secs]
4.706: [GC 4.706: [DefNew: 66607K->3275K(71680K), 0.0239610 secs] 66607K->3275K(708864K), 0.0240050 secs]
6.271: [Full GC 6.272: [Tenured: 0K->3326K(637184K), 0.0963900 secs] 21201K->3326K(708864K), [Perm : 10908K->10908K(16384K)], 0.
                                                                                                    0964500 secs]
10.205: [GC 10.205: [DefNew: 63736K->556K(71680K), 0.0130200 secs] 67063K->3882K(708864K), 0.0130650 secs]
13.272: [GC 13.272: [DefNew: 64300K->924K(71680K), 0.0195780 secs] 67626K->4251K(708864K), 0.0196150 secs]
51.580: [GC 51.582: [DefNew: 64667K->1579K(71680K), 0.0098410 secs] 67994K->4906K(708864K), 0.0098890 secs]
60.023: [GC 60.023: [DefNew: 65323K->1894K(71680K), 0.0114410 secs] 68650K->5220K(708864K), 0.0114870 secs]
66.373: [Full GC 66.381: [Tenured: 3326K->5347K(637184K), 0.1355620 secs] 63147K->5347K(708864K), [Perm : 12857K->12857K(16384K)]
                                                                                                    , 0.1356120 secs]
71.608: [GC 71.608: [DefNew: 63743K->591K(71680K), 0.0056480 secs] 69091K->5939K(708864K), 0.0056950 secs]
76.656: [GC 76.656: [DefNew: 64335K->865K(71680K), 0.0053640 secs] 69683K->6213K(708864K), 0.0054100 secs]
80.637: [GC 80.637: [DefNew: 64604K->1010K(71680K), 0.0054380 secs] 69951K->6358K(708864K), 0.0054820 secs]
84.799: [GC 84.799: [DefNew: 64754K->1244K(71680K), 0.0067490 secs] 70102K->6591K(708864K), 0.0067960 secs]
88.540: [GC 88.540: [DefNew: 64988K->1373K(71680K), 0.0077560 secs] 70335K->6720K(708864K), 0.0078030 secs]
92.418: [GC 92.418: [DefNew: 65117K->1539K(71680K), 0.0080650 secs] 70464K->6886K(708864K), 0.0081130 secs]
95.461: [GC 95.463: [DefNew: 65283K->1640K(71680K), 0.0087570 secs] 70630K->6987K(708864K), 0.0088040 secs]

4995.248: [GC 4995.248: [DefNew: 14542K->904K(15360K), 0.0062490 secs] 112796K->99346K(151936K), 0.0062930 secs]
4995.737: [GC 4995.738: [DefNew: 14600K->848K(15360K), 0.0051820 secs] 113042K->99420K(151936K), 0.0052270 secs]
4996.139: [GC 4996.139: [DefNew: 14544K->859K(15360K), 0.0061050 secs] 113116K->99591K(151936K), 0.0061470 secs]
4996.478: [GC 4996.478: [DefNew: 14555K->881K(15360K), 0.0061880 secs] 113286K->99792K(151936K), 0.0062310 secs]
4996.766: [GC 4996.766: [DefNew: 14577K->812K(15360K), 0.0054280 secs] 113488K->99866K(151936K), 0.0054930 secs]
4997.103: [GC 4997.103: [DefNew: 14508K->999K(15360K), 0.0058960 secs] 113562K->100053K(151936K), 0.0059390 secs]
4997.531: [GC 4997.531: [DefNew: 14695K->916K(15360K), 0.0053940 secs] 113749K->100158K(151936K), 0.0054390 secs]
4997.929: [GC 4997.929: [DefNew: 14612K->1021K(15360K), 0.0063230 secs] 113854K->100384K(151936K), 0.0063650 secs]
4998.318: [GC 4998.318: [DefNew: 14717K->806K(15360K), 0.0058900 secs] 114080K->100500K(151936K), 0.0059340 secs]
4998.653: [GC 4998.653: [DefNew: 14502K->945K(15360K), 0.0055210 secs] 114196K->100639K(151936K), 0.0055640 secs]
4998.948: [GC 4998.948: [DefNew: 14641K->841K(15360K), 0.0060110 secs] 114334K->100796K(151936K), 0.0060550 secs]
4999.238: [GC 4999.238: [DefNew: 14537K->854K(15360K), 0.0061250 secs] 114492K->100979K(151936K), 0.0061670 secs]
4999.519: [GC 4999.519: [DefNew: 14550K->855K(15360K), 0.0059880 secs] 114675K->101134K(151936K), 0.0060320 secs]
4999.921: [GC 4999.921: [DefNew: 14551K->796K(15360K), 0.0050630 secs] 114830K->101236K(151936K), 0.0051060 secs]
```

By summarising the CPU times for garbage collection in an interval and divide by the time elapsed for that interval, the garbage collector CPU utilisation is obtained. We can also calculate the mean heap size from the logs, and find the max heap size.

The GC-logging does not result in any measurable overhead to the application. The JVM keep track of these statistics anyways. The command line switches just ask the JVM to output the data to a log file.

2.8 Web application architecture

The following sections maps the software to hardware, so a description of the term node is in place:

Node: A node is a computer running some application or service. Depending on the context it means the whole system with both hardware and software, or just the bare computer hardware. The notion of a node is often used for computational computers in clusters.

The software of a typical web application is organised in tiers, where each tier has defined responsibilities. These software tiers can be mapped onto computer nodes in different ways, where each mapping configuration has advantages and disadvantages when considering performance, scalability and other quality requirements, such as quality requirements are fail-over for user sessions, high availability etc.

Web application architecture refers to how the application is divided in several tiers, and how the tiers are deployed on a cluster of nodes. The tiers are logical services, and it is the application developer's concern how to utilise the tiers. All tiers can run on a single node, or each tier can run on separate nodes. In more advanced configurations, even a cluster of nodes can be assigned to a single tier.

Web tier: The front end of the application, which the clients communicate with. The tier provides static content (simple, static HTML pages and images) This tier can be clustered over several nodes, and requests from the clients will be directed to the nodes according to some load balancing policy. Examples: Apache, Netscape Enterprise Server and Microsoft Internet Information Server.

Presentation tier: Provides dynamic content (t.ex. Java Server Pages). For small applications, the web tier is often integrated within the presentation tier software We call this kind of software as servlet engines. Examples: Tomcat, Resin and Jetty.

Object tier: Deals with java objects such as Enterprise Java Beans and RMI classes. Software that implements the object tier is often called an application server. The application server is actually an implementation of the J2EE specification. Examples: JBoss, JOnAS, WebLogic and WebSphere.

Only a subset of all possible configurations will be discussed here, choosing the most relevant according to what is often used in production environments. Choosing the configurations also depends on what can actually be configured on our available and very limited hardware resources. See [25] for more detailed description of different production evnironment configurations.

"Application server" or "app server" from now on means the hardware node that the server software runs on, or the server software, depending on the context.

2.9 Server topology

Some clarification is needed on what services are offered by the server components, and how they can interact with each other. The software stack is as follows:

Client Web browser –> HTTP server –> Web Servlet container –> Application server –> Database server

- A client web browser sends requests to a *HTTP web server* that serves static content.
- If dynamic content is needed, a request is sent to the *Web Servlet container*.
- If business logic is needed, a request is sent to the *Application server*
- If access to persistent data is needed, a request is sent to the *Database server*.

To clarify the various notations, table 2.1 shows the relation between different roles, tiers and the implementations used in this project.

Role	Tier	Implementation
HTTP server	Web tier	Tomcat built-in HTTP server
Web server (servlet container)	Presentation tier	Tomcat
Application server	Object/Application tier	JBoss
Database server	Database tier	MySQL

Table 2.1: Web application tiers and roles

These server components do not have to reside on separate computers, they all can run on the same computer, or some of them can even be integrated into another component:

- The Tomcat servlet container software comes with a built-in HTTP server. This is an adequate HTTP server when developing and testing the software, but in production environments separate HTTP server are often used to increase performance.[4]
- Application servers are often bundled with a web servlet container. In production environments, this is usually the best solution as there is less CPU overhead as a result of network messages between the web and application server.

2.10 Typical production environment

In a typical J2EE production environment, the application software would run on expensive servers, and even more expensive database servers are used as backbone. Usually there will be a cluster of HTTP servers to serve static content, and there would also be a firewall in front of both the web cluster and application cluster. The load balancer does what the name implies: balancing load between the HTTP or application servers.

When a user logs in, a session is created on the web server. "Sticky sessions" ensures that subsequent requests from a user are directed to the same web server as where the user data are cached. The load balancer and the web server are responsible for managing the sticky sessions.

In production environments the web servlet container is usually integrated in the application server software to remove the RMI overhead when sending network messages back and forth. See figure 2.4.

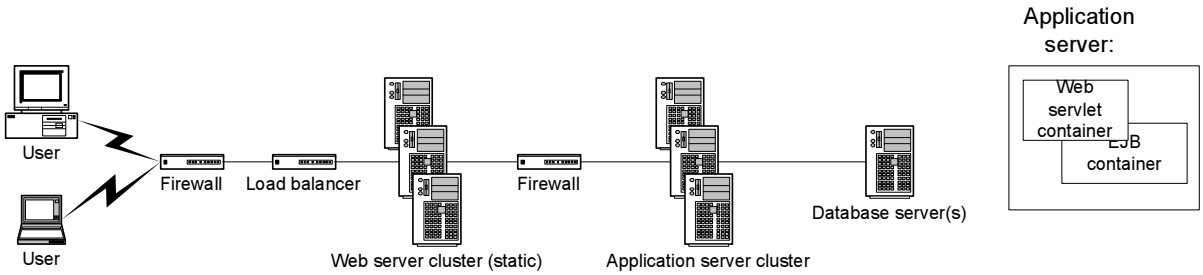


Figure 2.4: Typical web application configuration

[4]The built-in HTTP server is one of Tomcat's *connectors*. A connector is a piece of software that allows clients or other software to connect to Tomcat. Another Tomcat connector is the JK2 connector, enabling separate HTTP servers to communicate with the Tomcat servlet container. The JK2 connector is used as bridge between an Apache server running as load balancer, and the Tomcat web servlet.

3 Scalability

This chapter will cover the basis for the Scalability Assessment method, which will be presented in Chapter 4. For background material, see [3, 8, 4].

3.1 Introduction

With the growth of network based services, for instance web services, scalability has become a buzzword. Companies claim that their solutions scale well, but what do they actually mean by that? In this chapter we will define what we mean by scalability. This is necessary to be able to investigate scalability formally.

Scalability often becomes an issue when the number of users on a system are increased beyond what it was planned for. Aditionally, users may change their usage pattern, e.g they use some services more often than before.

3.2 Motivation for scalability analysis

One question arises: why bother with scalability analysis when hardware is cheap compared to man power? It should be just a matter of buying more or faster servers. The economical consideration is that history shows us that software requirements keep up with the hardware performance. Another aspect is that it may not be feasible to just add hardware to the problem if there are architectural limitations such as centralisation.

Enterprise applications can be very large and complex, relying on software products from many vendors. It is impossible to have a full overview of how these products behave in every situation, in which some situations can introduce unwanted performance penalties to the system. System architects have to make a lot of decisions when implementing enterprise applications, and they may not always be aware of the implications of their design choices.

The architectural design choices are often made on the basis of the current business situation, but a scenario may arise where the architect forgets (or deliberately ignores) to consider e.g a 100-fold increase in user base or transaction volume. Centralised code then can become a major bottleneck.

In the construction industry engineers model and simulate bridges or buildings before they are built. But this is less often the case in software engineering. Here follows a few reasons why one should consider scalability analysis in the development process:

- Systems are often large and complex, and it is hard to have full overview on how each component interact with the others.
- Changes in workload can have unforeseen consequences on performance. Even changes in the work definition (but not the load) can lead to higher load on software sub-components. This may result in queuing effects for resources that in turn depend on other stressed software sub-components.

- Some implementation choices can lead to bottlenecks in the system that cannot be remedied by faster servers. Man power is expensive, and it may cost too much or cause too much downtime to improve a system that is built on bad architectural choices.

3.3 Scalability vs. performance evaluation

The performance *P* of a system *S* is meaningless without a defined workload *W*, *see* [4]. *P* is a response of *S* to a stimulus of *W*, and can be illustrated with the functional representation:

$$P(S, W)$$

Consider a system under investigation with a given size. When modifying the workload, but keeping the system, we consider it a subject to performance evaluation:

$$P'(S, W')$$

On the other hand, if the system is modified to keep up with changes in the workload, we consider it a subject to scalability analysis:

$$P'(S', W')$$

Performance evaluation When the load is increased and the system remains unchanged, it is a pure performance evaluation issue.

Scalability evaluation When the load is increased, and the size of the system is increased to cope with the new load requirements, it is an issue of scalability analysis.

From a business point of view it makes sense to mix the concepts of performance and scalability. A CEO may ask: "what happens when our new marketing plan works, and we get twice as many customers - is our current system able to scale well and handle the increased load economically?". When investigating and exploring scalability on the other hand, clear distinctions must be made to be able to approach the problem in a sound manner.

3.4 General scalability definition

A simple definition of scalability is a system's ability to handle growth economically. For scalability analysis and research, a more specific definition is needed. Peter Hughes and Gunnar Brataas use this definition in [8]:

> "An architecture is scalable with respect to an IT profile and a range of desired capacities if it has a viable set of instantiations over that range"

An IT profile denotes all other requirements than capacity, such as functional requirements and performance requirements. A viable set of instantiations means that each instantiation is feasible both in a business perspective and in a technical perspective. Bottom line, scalability is defined in the context of requirements.

3.5 Work and load

A clear distinction is made between work and load. The notion of workload is separated into functional work and the dynamical load. Work specifies what operations are invoked on a software component, while load is a measure of how much work the system is supposed to process.

Operation An operation is a service provided by the system at some level. At the highest level operations can be HTML requests available to a client user, such as logging in to a system. At lower levels operations can be methods in software classes, or even read or write requests in hardware components.

Work Work is what we (or system components) request from the system, defined by the operations provided by the system. Work can be expressed at any level of the system, but is mostly attributed to what a client user can access. It specifies what operations are invoked on a software or hardware component.

Work-mix Work-mix specifies the relative frequency between operations in the work definition. Work-mix is expressed as a vector with one element for each work operation, such as $[\,1\,login,\ 3\,newpayment,\ 3\,paymentconfirm,\ 1\,logout\,]$

Load Load specifies intensity or the rate of incoming work-requests. This can be the arrival rate of requests, or arrival rate of users logging in to the system.

3.6 Size vs capacity

Capacity describes how the system performs with respect to a given workload. But capacity is not always equal to the raw specifications of the physical resources. Therefore we refer to the physical resources as size. As an analogy, consider how the tables are arranged in a restaurant. The size is the area of the floor, and capacity is the number of people that can be seated. A RAID1 mirroring solution is a good example: The raw storage size is two disks, but the capacity is only one disk since both disks keep the same data.

Size The size is a metric of the potential of a physical resource. It denotes the raw processing power, raw disk storage or raw bandwidth, depending on what physical resource it is.

Capacity The capacity of a system or device is a metric of how the potential of a physical resource is utilised under a given load for given quality requirements.

Consider CPU processing. The size metric reflects how powerful a system is compared to the baseline system, where the baseline system has size = 1. The size of an upgraded system is quantified as a factor of the processing power between the upgraded and the baseline system. When size = 2, it reflects that the upgraded system has doubled processing power compared to the baseline.

Note that the definition of size and capacity is valid for the three dimensions, namely processing, storage and connectivity. The previous example dealt with processing power, but the example is also valid for storage capacity of a disk, or the bandwidth capacity of a network interface.

The size and capacity of the system is related to the type of work, since different work can yield different sizes of a single system. For scalability considerations the size is defined with respect to one work-mix, and that work-mix definition will stay fixed throughout the whole scalability analysis. See Section 8.3 for the work-mix definition used in this project.

3.7 Scalability function definition

With the notion of size, capacity and work-mix, we can define scalability of a system in a more formal way. The following definition is found in [3]:

> "In its simplest form, scalability describes the degree to which the size of a system has to be increased in order to achieve a desired increase in processing capacity. For a given workmix w and system S, this relationship may be described in terms of a scalability function $C_{S,w}(k)$ relating capacity C to relative increase in size k"

A scalable system will have a capacity C which is linearly dependent on k. The system must be scalable over a given range of load.

A poorly scalable system will have a capacity C which depends on powers of k less than 1.

3.8 Dimensions of scalability

As mentioned in the last section, physical resources have different metrics depending on what resources they are. One dimension is needed for each metric, and Peter Hughes and Gunnar Brataas identifies three dimensions in [8]: *processing*, *storage* and *connectivity*. Note that we also use these dimensions for software components. The capacity is defined with respect to the system as a whole or on individual subsystems at various hierarchical levels. Examples of capacity are given with respect to the three dimensions:

Processing capacity Processing capacity is the rate at which specific work can be performed. On the system level, one example of capacity is the number of active users versus the number or size of processing nodes. On subsystem level, examples of processing capacity are CPU processing power, disk service rates and network bandwidth.

Storage capacity Storage capacity describes the amount of available storage at some level, such as number of accounts or products that are possible to store in a database, or effective disk storage on file servers. The storage capacity relates how much you can store versus the raw storage space needed.

Connectivity capacity Connectivity capacity describes the effective number of access points to a system or subsystem, such as total number of users that can connect to a web server or total number of connections to a database server. The connectivity is often constrained by threads, file descriptors or tables in the operating system or software.

3.9 Scaling function

The core in a scalability analysis is to answer what happens when the load changes over time, and when the system size is increased to cope with the new load. This is visualised in Figure 3.1. The scaling function maps size on to capacity in a particular scaling dimension. The dimension is either processing, storage or connectivity. The shape of the line determines the relationship of size and capacity: A straight line represents linearity, while a slope is denoted super-linear or sub-linear . Linear scalable systems are the ideal cases, where capacity is proportional to the increased size under some defined workload.

The steps in the figure denotes that a system is upgraded in discrete steps, since nobody buy a new and slightly better processor each day. Each step represents the actual capacity of the system for that

upgrade. At least for large values of the size, it is likely that a system behaves in a sub-linear fashion as shown in Figure 3.1.

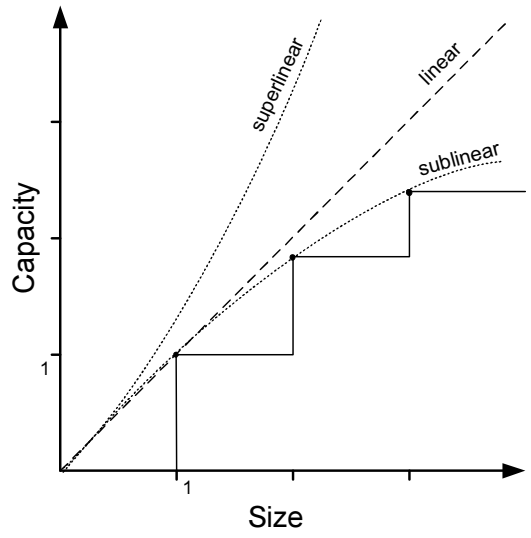


Figure 3.1: Figure 3.1The scaling function

The super-linear case is rare in systems as a whole, but can appear on subsystems or devices. One example of super-linear behaviour can be observed on SCSI disks with a high load of random requests. The disk controller sorts the requests, and some of the requests may be served in bursts rather than as single and random requests.

3.10 Hierarchic scaling

A system may be improved in two ways: The system's devices can be upgraded, or they can be replicated. Figure 3.2 shows how the system can be improved. Figure is taken from [8]. An upgrade on level II may be carried out by either replicating or upgrading the devices at level III.

As an example, consider a system with one CPU. This system can be upgraded by:

- Upgrading to a faster CPU
- Replicating the CPU by installing a dual-core CPU.

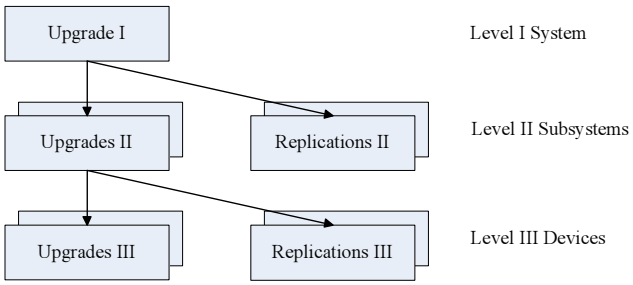


Figure 3.2: Three hierarchical levels of scaling options

Platform The platform is what the system runs on. It includes the hardware and some or all software, the boundaries depends on the point of view. One useful perspective is to consider both hardware and the operating system as the platform, viewing the operating system as the interface to the hardware. [1]

3.11 Non-linear sources

Development or deployment tradeoffs might introduce non-linear effects in a system. Regardless of how they are introduced, non-linearities must be taken into account for systems with evolving load. Chapter 11 describes the search for non-linearities in the context of the case study.

Predictable non-linearities: When a system is implemented, the developer often has to choose among several algorithms. These choices can introduce non-linear effects. Given limited hardware resources or requirements that are contradictory to one another, linear scalability may not be obtainable, and the developer has to choose where to introduce non-linear effects. Then the architect must consider where the system can afford to have sources to non-linear effects, according to some business plan of how the load is going to change.

Unpredictable non-linearities: Scalability issues can be dangerous when they are not predictable. An example of this is when the developer introduces bugs in the implementation of an algorithm, and the result may not be discovered until production phase of the system, or even not until the system is used in a particular way years after release. Another possible source of non linear effects is that the developer designed a flawless system, but the use of the system changes over time, and the design does not support the changes in a flexible way. Several items can also interact in such a way that scalability becomes unpredictable.

3.12 Non-linear effects

Several factors may introduce non-linear effects to a system, and these are the main groups of non-linear effects:

[1] For instance, the Java Virtual Machine software emulates a generic hardware platform, and from a developer's perspective the JVM represents both hardware and an operating system.

Congestion effects When several processes compete for resources, congestion effects occurs. The response time will increase exponentially when the system is saturated. These effects are dealt with by performance analysis. In SAM, contention effects are eliminated by operating the system under equal contention levels. The term for this is the "operating point", and is described in Section 3.13.

Software effects These are effects introduced as a result of architectural choices and implementation of the software. The effects are mostly workload- or dataload dependent, such as centralised code and increased data volume for each user.

Platform effects Platform effects may occur when replicating systems or subsystems. The increased resource usage may be a result of overhead in maintaining consistency of the data over replicated systems.

3.13 Operating point

Congestion effects are removed from the analysis by maintaining an operating point for a system. The operating point balances the throughput and responsiveness of a system.

The operating point enables us to compare systems with different scale factors under equal conditions. It reflects a fixed point with regard to some requirements. The requirements can be *equivalent utilisation* or *equivalent response time* for the system as a whole. See [1] for further details.

Equivalent utilisation The system is utilised equally for all scale factors. The practical implication is that the node with the bottleneck device is kept at the same utilisation when increasing the scale factor. This is done by adjusting the load when the system is upgraded. The result is that the response time is (in ideal cases) halved when doubling the system. It is a conservative approach, and one may criticise equivalent utilisation from a business point of view: When the system is scaled up by a factor of 2 and the response time is halved 2, the system has underutilised resources compared to the initial case. Such criticism is only valid when the initial response time was in fact sufficient.

Equivalent response time For this operating point, the response time for a transaction is kept constant for all scale factors. This approach yields better scaling than equivalent utilisation. The downside is that the approach only supports a limited increase for the scale factor. [2]One example of equivalent utilisation is the approach used in this project. The operating point is set to a CPU utilisation of 65%. When upgrading the system, the load is increased until the upgraded CPU is also utilised by 65%.

3.14 Scaling consistency

Systems can be scaled up in the three scalability dimensions. For each particular dimension, subsystems can be scaled up as well. This gives a lot of possible scaling paths. The subject is explained in [8], and is formalised in [3]. Note that we use the term load for both workload, dataload and connectivity.

3.14.1 Scaling with respect to load requirements

Scalability is analysed with respect to a set of load requirements, where a load requirement is defined for each of the three scaling dimensions. The load requirements evolve over time, and systems are

[2]For an initial utilisation of 60%, a scale factor of 4 would lead to a utilisation of 90%. See [1].

scaled to meet the new requirements. In [8], the relationship between the load, operating point and capacity is defined:

> "The scale is defined with respect to a reference load at time t=0 running at a defined operating point on a reference system. Without loss of generality, we take this load to define also the capacity of the system in each dimension at the required service level "

There are three different ways to scale the requirements:

Strict scaling The work-mix is kept constant. Scaling requirements are changed by the same factor in all dimensions.

Differential scaling The work-mix is kept constant. Scaling requirements are changed by different factors in the three dimensions.

Work-differential scaling The workmix is changed. The load may be changed in one or more of the dimensions.

3.14.2 Platform scaling path

The platform resources are increased to meet the new load requirements. We consider scaling of the subsystems in a particular dimension.

There are two ways to scale the platform resources:

Uniform scaling The scale factor is changed by a constant *k* for all devices in all dimensions.

Non-uniform(skewed) scaling The scale factor is changed by different values of *k* for each dimension. *k* is a vector k = [k_{proc}, $k_{storage}$,k_{conn}]

3.15 Baseline and upgraded system

Baseline system The reference system at time t=0, whose size is 1. The baseline definition includes hardware, software and configuration parameters. Strictly speaking, the baseline also holds the operating point by sustaining a given load.

Upgraded system The baseline system is upgraded to meet the new load requirements at time=T. Upgrade is done by device replications or device upgrades .

3.16 Operational analysis

Scalability analysis requires knowledge to some basic operational quantities and laws. See [4] for a more detailed description. The subscript $_i$ indicates a component in the system, while the subscript $_0$ denotes the system as a whole. Consider a system measured over a fixed time T. The qantities are:

Number of arrivals A_i, Busy time B_i and number of departures C_i

From these quantities we derive the other:

Arrival rate $\lambda = \frac{A_i}{T}$, Throughput $X = \frac{C_i}{T}$, Utilisation $U = \frac{B_i}{T}$, Mean service time $S = \frac{B_i}{C_i}$

Combining the *utilisation law* $U_i = X_0 S_i$, and the *forced flow law* $X_i = X_o V_i$, we get the *service demand law* $U = X D_i$, where the *service demand* is $D_i = V_i S_i$. The service demand law is used to calculate the resource usage per work unit put on the system as load. Capturing the CPU service demand is a simple recipy of three steps:

1. Measuring the mean CPU utilisation over time T.
2. Counting the number of departures of work units from the system in that period. Calculate the throughput.
3. The service demand is calculated by dividing the utilisation by the throughput.

With the service demand, we can calculate the maximum throughput on a device. Max throughput is obtained when the device is fully utilised, $U_i = 100\% = 1$. Maximum throughput is then: $X_0 = \frac{1}{D_i}$.

The service demand D_0 denotes the total service time that a work unit needs on all resources on a system. The total respone time of a work unit is the total service demand plus wait time on all resources.

4 Scalability Assessment Method

The Scalability Assessment Method (SAM) is introduced in this chapter. SAM addresses the task of analysing scalability of a system's architecture, whether the architecture is implemented or not.

SAM is a method developed at IDI, NTNU by Peter H. Hughes. The method deals with all aspects of a scalability analysis. SAM is about analysing and evaluating scalability of an IT-system in a formal way. It shows how to scope the investigation, what hierarchical considerations are to be made, how to build and analyse models and how to use the models to explore scalability. The document [1] describes the key considerations and limitations of the analysis.

First we describe method of the classical modelling cycle, and then we describe how SAM differentiates from the classic method.

4.1 Modelling overview

The method presented here is the classical modellling cycle. The overview serves as a bootstrap to the more complex SAM model in following chapters. Figure 4.1 relates the different tasks in SAM to each other. The baseline system and model are denoted S_0 and M_0, while the upgraded system and model are denoted S_1 and M_1.

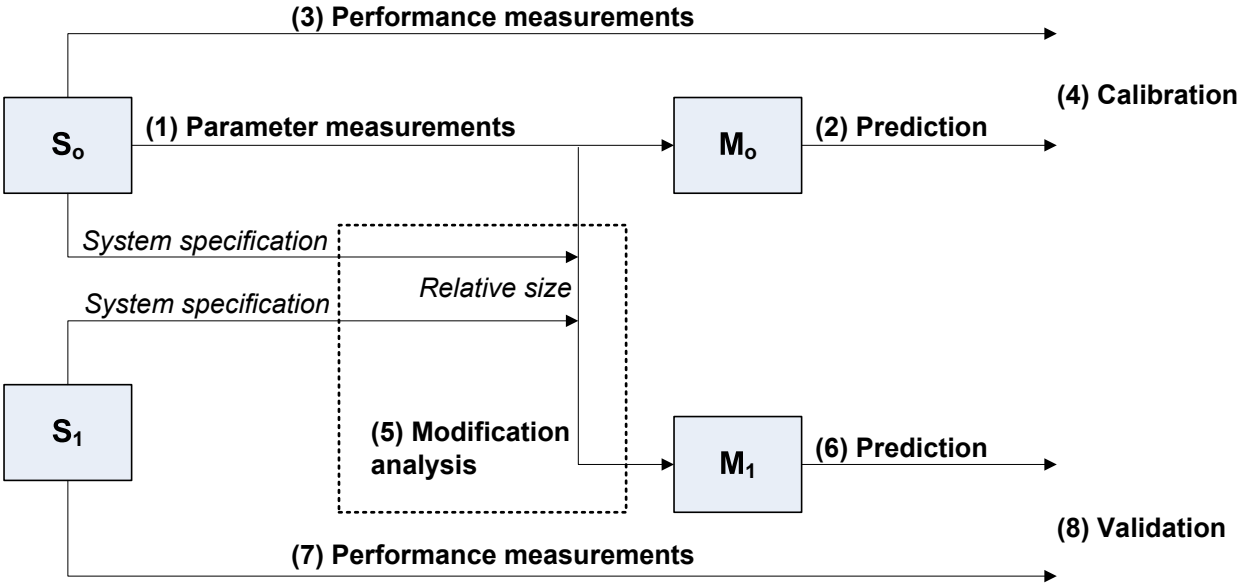


Figure 4.1: Modelling overview

Below are the three classical stages in the modelling cycle presented with respect to SAM:

Construction stage In the construction stage, a performance model $M_0$is built and parameterised with parameter measurements(1) on the initial system S_0. The output from the model (2) is compared to the performance measurements (3) from the same system. If there are only relatively small differences between the system and the model, the model is calibrated by adjusting the model parameters. For a given workmix, the model should now yield the same performance as the system.

Projection stage In the projection stage, a modification analysis (4) is performed to build a model of a modified system. Based on the system specifications of system S_1 relative to S_0, we build a model $M_1$to reflect the new system.

Validation stage In the last stage, the output(6) from the model M_1 is validated against performance measurements (7).

Depending on what state the software is in, modifications to the analysis might be required:

- Development phase: The application is not implemented yet, and it cannot be measured in its entirety. A baseline model must be built from any available pilot measurements, from code inspection or experience from best practices.
- Testing or production phase**:** The system is implemented and running, and the baseline can be measured.

The analysis must also be modified depending on the availability of hardware:

- If both baselineS_0 and upgrade S_1system does not exist, the models must be built from code inspection or experience. No validation can be performed.
- If the only the baseline system S_0exist, the validation stage must be left out.

4.2 SAM procedure

The SAM procedure is actually more of an iterative process rather than the top-down-approach presented in the overview section.

The key difference between the classical modelling method and SAM is the feedback loop. The feedback loop ensures that the system measurements or model predictions are fixed to a chosen operating point, where the load is increased or decreased until the system reaches the operating point. Each time we change the scale factor, the load must be adjusted to reach the operating point.

Figure 4.2 illustrates the experimental procedure in more detail. We list and describe the steps in the procedure:

1. **The work-mix** is fixed throughout the entire analysis, i.e. it is fixed for all values of k. The work-mix is represented as a vector, with one element for each top-level operation.
2. **The scale factor** $\overrightarrow{k}$ is a multi-dimensional vector, comprising processing, storage and connectivity. The scale factor represents the size of the system compared to the reference system at time t=0 (the baseline).
3. **The system model** takes input from either component models or measurements. This depends on the granularity of the models. A component model is represented as an SP model in SAM (see Section 4.5 for a description of SP), and the system model is represented as a queuing network.
4. **Validating the baseline** is a task where the output from the system model is compared to system level measurements. The baseline model is considered validated if it yields the same results as measurements, within a satisfactory error margin.

5. **The modification analysis** is a task of modifying the model parameters to project an upgraded system. The result of the analysis is represented as a function $f(\overrightarrow{k}, \overrightarrow{L})$. The function depends on the scale factor $\overrightarrow{k}$ and the load $\overrightarrow{L}$.
6. **Validating the prediction** can be performed if the upgraded system exists. Output from the projected model is compared to the system level measurements on the upgraded system.
7. **The capacity metric** is the load that the system can sustain at a fixed operating point. The capacity metric is the $C_{S,w}(k)$ function defined in Section 3.7.
8. **The gain vector g(k)** is the capacity gain for a scaling vector $\overrightarrow{k}$. The gain is found in each dimension by dividing the capacity metric for the upgraded system by the baseline system.

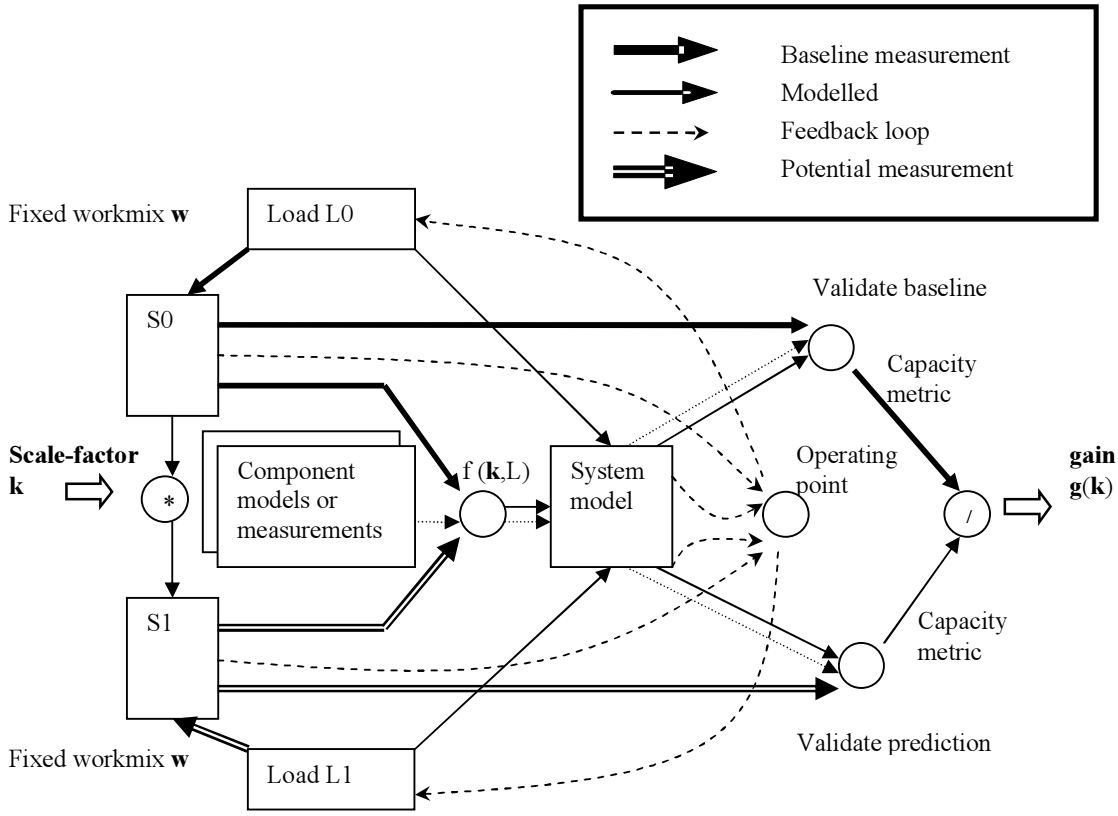


Figure 4.2: SAM procedure. Figure by Peter Hughes, 2006

4.3 SAM main steps

This is a summary of the main steps of SAM. We will use this list when applying SAM in Chapter 5.

1. Define scaling objective (scope, scaling paths and invariants)
2. Establish baseline
3. Construct and parameterise the static model (component model)
4. Construct and parameterise the dynamic model (system model)

5. Execute the dynamic model to obtain resource usage on hardware devices
6. Measure the baseline system to calibrate(and validate) the baseline model resource usage
7. Explore scalability by identifying feasible scaling paths (identifying scale factors)
8. Make projections to the baseline (modification analysis)
9. Validate the projections by measurements

4.4 Modelling work and load

The SAM approach requires the use of two different modelling paradigms: static and dynamic modelling. The static and dynamic model are combined to produce performance estimates of a live running system. Static modelling captures the static aspects of the system, and dynamic modes captures the contention aspect. A baseline model refers to the combination of the static and the dynamic model.

We present SP modelling and then a brief description of dynamic modelling. Finally we show how to combine the models to represent the baseline model.

4.5 Static model: SP

A central modelling technique used in SAM is the Structure and Performance specification (SP). The SP specification is described in [5]. SP addresses the problem of maintaining an overview of the resource usage in complex system.

SP maps resource usage for top-level operations to hardware devices. In an SP diagram, hardware devices are at the lowest level. Software modules are aggregated into *components*. A component or device offers services, or *operations*, to components on higher levels. The resource usage of the system is captured as *complexity functions* in matrices. The matrices represent interaction between the SP components.

Component Software modules or classes are aggregated into logical components.

Operation A software component offers a set of operations, or services, to a higher-level component.

Top-level operations are offered to the client user by the system seen as a whole. They serve as entry-points to the system.

SP represents the static view of the system. The purpose of the static model is to ultimately estimate the devolved work on the hardware devices in terms of service demands.

Devolved work The output of an SP model is devolved work. This is a unit-less metric that indicates the resource usage on the hardware devices for each class in the work-mix. In the dynamic model the devolved work is combined with hardware device speeds which produces service demands. Devolved work can viewed as CPU instructions.

The system is modelled as a directed as a non-circular graph. Figure 4.3 shows an example of how a software component is mapped to hardware devices in SP. This instance of the model represents a single node, where all software is composed into a single software component. To represent processing between two components we use a thin line. The thick line is used for storage and a the dotted line is used for communication. Note that all software components in SP must have a processing link to CPU.

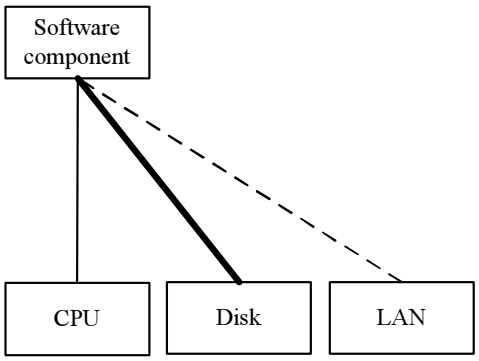


Figure 4.3: SP

Figure 4.4 shows how the complexity matrix relates to the components. The operations in component A is mapped to the operations in component B in the complexity matrix. Resource usage in the static model is captured as complexity functions in the complexity matrices. The link between each component in SP is represented by a complexity matrix.

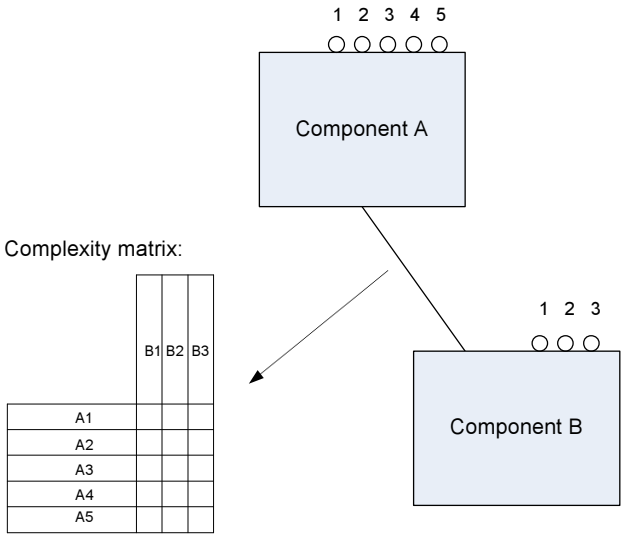


Figure 4.4: SP complexity matrices

4.5.1 Dimensions of SP

The SP dimensions are not the same as the scalability dimensions. The scalability dimensions denotes capacity in each dimension, while SP dimensions denotes the work between SP components.

There are three dimensions in SP, namely processing, memory and communication:

Processing This dimension is about processing in some form, such as CPU processing, disk processing, or the processing in a software component. Each SP component has to be connected to a processing unit (CPU) with a processing link.

Memory Data is passing by in the memory dimension. The metric for the memory dimension is storage units, such as kilobytes or megabytes. Memory denotes the size of the messages that are sent between components. An example of memory is the data (or size of the data) sent to a disk subsystem.

Communication The communication dimension is memory that is distributed, meaning that data is distributed over physically disjoint components. A communication example is the size of the data that are sent to a network subsystem.

4.5.2 Types of complexity functions

This section describes the complexity functions in an SP model. Measurements and estimations are costly, so it is important to decide which dependencies to capture. In [6] three possible types of variables and two types of complexity functions are identified:

Type	Controlled when	Controlled by	Dependence
A	Installation time	Software version, installation parameters	
B	Run time	Scale factor and scaling policy	load, dataload
C	Uncontrolled factors	Eliminated by appropriate randomised design	sequence of operations, burstiness

Table 4.1: Variable types in a static model

Type		Controlled by	Manifested as	Note
1	Constants	Type A variables	0,1,k	1)
2	Functions	Type B variables	n'th order polyminal, log	

Table 4.2: Complexity functions in a static model

1) Note that k may be a function of type A variables.

4.5.3 Dependent/independent properties

Variables may depend on properties of the system. Otherwise they are called independent. The variables are either dependent or independent on at least one of these properties. A postfixed "D" means dependent, and "I" means independent. Three main groups of dependent properties are listed: The acronyms is then LI/LD, DNI/DND, and DSI/DSD.

The "load" (L) property is well known, a metric that indicates the rate of work received by the system or software layer .

The "data name" (DN) property means *what* data are being requested. For example when specific users are requested, their data may or may not exist in cache for a given software layer. This property mainly effect cache hit-rates.

The "data size" (DS) property on the other hand means *how much* data are being requested. An example is how much data needs to be transferred per user between the database and application server. This affects memory utilisation, local area network utilisation, or software layers as the garbage collector routines.

4.6 Dynamic model

The dynamic model captures the dynamic aspects of the system. When several entities compete for limited resources, contention will arise, resulting in higher response times. If the system is over-utilised, thrashing can occur, and the throughput will drop.

We choose to model the system as a queuing network, which enables us to introduce load to the analysis.[1] A queuing network is parameterised by the arrival rate that jobs (requests) enter the network, and service times for each queuing centre.

4.6.1 Queuing networks

From basic queuing theory we know that when many clients want to use a resource, they are put in a queue waiting to be served. A queuing network represents the contention for resources in the system.

A queuing network is defined by two main properties:

- **Load:** arrival rate of requests to the system as a whole, or number of clients accessing the system. This depends on whether the system is open or closed. See Chapter 14 for a discussion of closed versus open queuig networks.
- **Service times** for each queuing centre.

The physical hardware devices (CPU, disk, LAN) are represented as queues in the queuing network, with one queue for each device. A queuing network can itself consist of many smaller networks. Inside of the system a customer may have to visit several queues before he can leave.

We assume that the capacity (buffer length) is infinite, and a FCFS dicipline (first-come first-served).

4.7 Baseline model: static and dynamic model combined

The baseline model comprises of both a static and a dynamic model.

Figure 4.5 shows how the static and dynamic model relates to each other. The workload is separated into functional work and dynamic load.

The static model maps top-level operations to low-level operation on the hardware devices. Executing the static model gives us the devolved work for all classes in the work-mix. The devolved work is used as input parameter to the dynamic model.

In the dynamic model, the the devolved work is combined with the performance specification of the hardware device, which gives us the service demand. The service demand is used to parameterise the service centres in the queuing network. The arrival rate of each class in the work-mix is used as input to the dynamic model. The dynamic model produces performance estimates when executed.

Executing the baseline model consist of all steps described in this section. Depending on the presence of run-time dependent variables in the static model, the statis model may be run once or every time the load changes:

- If there are no run-time dependent variables in the static model, it is sufficient to run it once, and then re-run the dynamic model for each change in the load.

[1] Other options are to model the system as layered queuing networks, or simulate the system if a more detailed dynamic model is needed.

- In the presence of run-time dependent variables in the static model, the model gets a bit more complicated. Dynamic load properties is used as input to the static model's complexity matrices. Both the static and dynamic model must be executed for each change in the load variables.

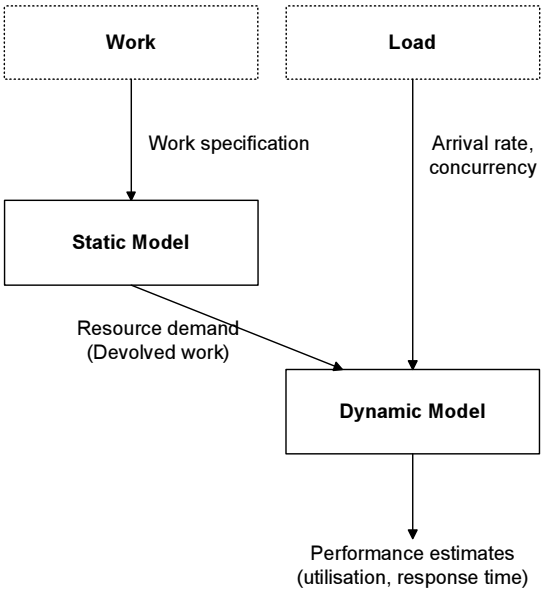


Figure 4.5: Baseline model comprises of static and dynamic model

Part II

Case Study

5 Applying SAM

We presented SAM in Chapter 4. We are going to apply SAM to the case study, and we present a brief outline of the work performed in the case study. This chapter acts as an index where we refer to the chapters where the practical steps are performed.

First we describe the context for the project and how it influenced this project.

5.1 Context basis

The research context presented in Section 1.6 influenced this project in two ways. First we recognized the need for SAM infrastructure by reviewing the work of Fagerlie and Landmark. Having an infrastructure has several advantages:

- For modelling the system, pilot measurements are needed to get an idea of scope and system constraints. This affects the modelling detail and granularity.
- To measure a system, technical support is needed to aid the configuration and deployment process. The motivation is to be able to save deployment time and be able to perform more experiments in a given project time frame.
- To get reliable results, there is a need for controlled experiments. Experiments should also be cheap in terms of man-hours, enabling the experimenter to test various aspects of the system and perform a lot of experiments.

Secondly we assumed there were non-linear effects in the system that we could measure. Before we were able to investigate the effects by measurements, a lot of work went into detailed modelling of the system, since the presence of non-linear effects requires more detailed models. As we did not find any significant non-linear effects, the detailed SP modelling work lost some value in the light of this project. This affects the consistency of the report, where we describe modelling steps while we do not actually perform the steps in the case study:

- The description of how to build and manipulate SP models does not match the simple SP model we use in the case study. The SP model can be omitted and we only need to build a system level model. Even though there were no measurable non-linear effects, it does not mean that there are none. The work on SP modelling are therefore kept, and so are some of the discussions around modifying SP models. The early work on detailed SP modelling can be found in Appendix G.
- The report describes how to discover different types of complexity functions, but little work is done in the case study on the subject.

5.2 The phases of SAM

We present the procedure of SAM as a broad serie of phases. The phases are derived from the SAM procedure and main steps in Section 4.2 and Section 4.3.

5.2.1 Establishing baseline

The first phase sets the ground for being able to perform measurements and model the system.

- Chapter 6: We present the hardware and software to be used in the case study. The available hardware and the software we choose put constraints on what we can analyse.
- Chapter 7: The scaling objectives and scope for the analysis are set.
- Chapter 8: We model the workload with respect to the available client services offered by the BankApp application software. The workload is modelled as we think an average real-life user would use the system.
- Chapter 9: The script toolbox developed in this project is presented. The scope, limitations and extensibility is evaluated.

5.2.2 Modelling baseline

In these chapters we make the baseline model of the system. To obtain the parameters, we measure the system by component or system level measurements.

- Chapter 10: Pilot measurements are performed to get an idea of the *scope and detail* of the model.
- Chapter 11: We search for non-linear effects to decide the *granularity* of the model.
- Chapter 12 and Chapter 13: The SP model is constructed and parameterised. We obtain the parameters analytically and by measurements.
- Chapter 14: The dynamic model is constructed and parameterised. We use parameters from the executed SP model as input to the dynamic model. The load is altered until the model reaches the operating point. Finally we validate the baseline model against system measurements.

5.2.3 Scalability of the upgraded system

In the last phase we project an upgraded system to the baseline. We will then measure the upgraded system to validate the projected model.

- Chapter 15: The scale factor is determined by measurements. We perform a modification analysis to project the upgraded model. The projected model is then validated against measurements. The last step of our scalability analysis is to compare the capacity of the baseline and the upgraded system.

6 Platform and software

This chapter will introduce the software and hardware used for measurements. The available hardware resources limits both the possible measurements and the possible validations of a scalability exploration.

6.1 Baseline configuration

Figure 6.1 shows a the baseline configuration. Each server run on a separate node. Several nodes run a load generator that emulate users submitting requests to the system. Each load generator node can represent hundreds of users.

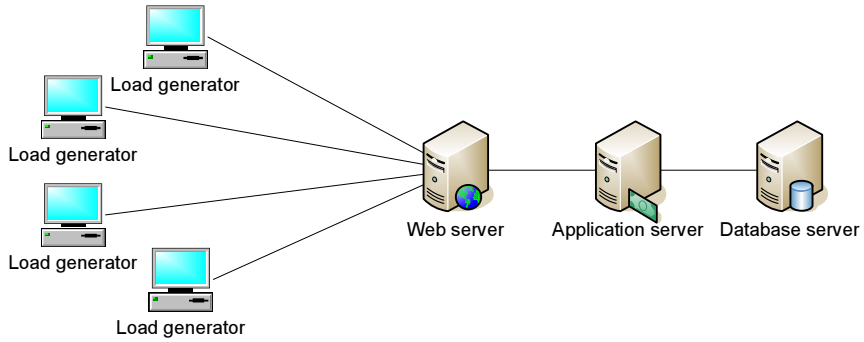


Figure 6.1: Baseline configuration

This setup is a simplification from Figure 2.4.

We have made some simplifications according to the scope of the project. If it can be assumed that a component only adds a load-independent delay to the total response time, that component may be ignored for scalability analysis purposes.

- Firewalls are disregarded since they impose only a tiny delay to the response time when they are properly configured and sized. In any case, firewalls are expensive hardware devices and they require a dedicated hardware environment.
- The web server handles both static and dynamic content. Separate HTTP servers are not needed, since there is little static content to be served from BankApp. There are only a few pages in BankApp, where none of them are pure static pages, and there are only two tiny images to be loaded. In any way, static content can be served from virtually any computer, it is just a matter of pointing to another computer in the html file.
- The web server runs on a separate node. It is easier to deploy applications on an integrated web and application server, but for parameterising purposes we want to separate the effects that each component (web and app) has on the hardware. This will lead to quite a bit overhead

in the messaging between the servers, but the reward is that we are able to obtain separate parameters for the web an application server. When running an integrated solution, a profiler must be used to separate the web and app server effects. Obtaining parameters by profiling is considered further work.

Measurements revealed that the web server was utilised roughly 3.5 times more than the app server. To be able to measure higher load on the app server, a modified baseline was configured. Using the Apache HTTP server as a load balancer, requests could be forwarded in a round-robin scheme to the web servers. When sharing the load on three or four web servers, the application server could be measured with a much higher utilisation. The modified baseline is useful for parameter measurements, when considering only the application server. Figure 6.2 shows the modified system with a load balancer and four web servers.

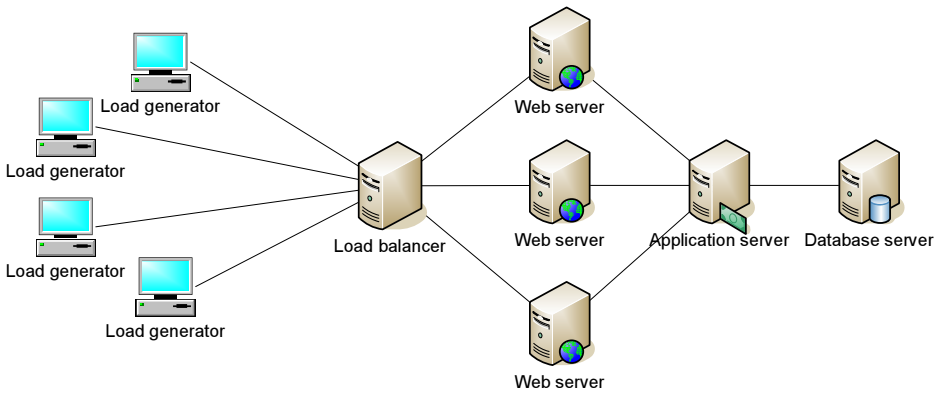


Figure 6.2: Modified baseline for increasing load on application server

6.2 Clustis2

Measurements are performed on Clustis2, a computational cluster of the Division of Intelligent Systems (DIS) and the Division of Complex Computing Systems (KDS) at IDI, NTNU.

The cluster consist of a master node and 20 computational nodes, interconnected by gigabit network interfaces. The nodes are protected from the Internet behind the master node, which provides a secure intranet environment without having to consider security issues. Through NFS mounting, the user's home directory can be accessed from all nodes.

Since Clustis2 is a shared resource for many users, major software and hardware modifications to the system are out of scope. The average Clustis user has very different needs from what a scalability measurement user has. Average Clustis users are mostly interested in a homogeneous node environment for performing complex computing and other distributed tasks. They do not need the interactivity and specific hardware configurations.

6.3 Old nodes on Clustis2

ClustIS was the first cluster we used on IDI for scalability measurements. It had 30+ computational nodes, with AMD CPU's (MP1600+ and XP1700+). The first measurements were performed on this cluster. Later, IDI bought a new cluster called Clustis2 with Pentim P4 nodes (3.4GHz)

The new Clustis2 nodes could represent upgraded nodes in the scalability analysis, so measurements were performed on both ClustIS and Clustis2. No need to say, a lot of work had to be done to synchronise the measurement scripts and files, as they evolved on the way.

The system administrators agreed on moving a few nodes from the old ClustIS to Clustis2. This way the old nodes could represent the baseline, and the new nodes on Clustis2 represent an upgraded system.

Unfortunately, the baseline nodes were taken offline before the end of this project. The measurement design and some scripts were improved afterwards, and the measurements could not be performed. The implications of this are dealt with in its respective places. See Section 10.9 and Appendix,Section C.1.

6.4 Available nodes

Clustis2 consists of 20 computational nodes. In addition, 5 computational nodes from ClustIS are available on the cluster, and are used as baseline nodes:

Type 1: ClustIS nodes (baseline nodes)

- AMD athlon MP1600+ (1.4GHz clock speed)
- 1GB of RAM
- 100Mbit network interface

Type 2: Clustis2 nodes (upgraded nodes)

- Intel Pentium4 3.4GHz
- 1GB of RAM.
- 1Gbit network interface

6.5 Software

The various software components are discussed below, specifying the version used in the measurements.

6.5.1 Linux operating system

Clustis runs a Redhat distribution with a Linux kernel Linux kernel 2.4.21-15.EL. This is a patched version with the new O(1) kernel scheduler feature that removes a lot of overhead when managing a large number of threads (hundreds or thousands). This is ideal since the Tomcat server nodes may use one thread per user session for some configurations.

The cluster is managed with the ROCKS cluster distribution, which comes with nice statistics features for the nodes. Some of these statistics are used in Section 10.7.

6.5.2 Java Virtual Machine

SUN's JVM version 1.4.2_06 was chosen as JVM. It is relatively well documented and accessible. The IBM JVM is said to be more resource effective in production environments, but the scope is to investigate JVM properties in general, not to tweak the system to run as fast and effectively as possible.

Java programs use preallocated main memory, assigned as heap size. For baseline measurements, each JVM instance running Tomcat or Jboss server are instructed to assign 350MB of RAM to the heap. For upgrade measurements, the heap is increased to 700MB. In some parameter measurements, these values are altered to measure heap effects.

The SUN JVM provides four garbage collectors, but for simplicity the standard garbage collector is used in all measurements. The focus is to measure the effects of garbage collection, not tweak it.

6.5.3 Tomcat servlet container

The Jakarta Tomcat version 5.0.28 is used as HTTP and web server. This is a standalone version of the server, not the Tomcat that is shipped as an integrated web server in JBoss application server.

The server are run with standard options. HTTP persistent connections are disabled by setting the keepAliveRequests property 1. The property limits how many requests the connection is kept alive for before the connection is closed. This means that a Tomcat thread will not wait for new requests from the last client, but will close the connection and accept new requests from any client. This way the number of Tomcat threads stays below 100. If we were to use keepAliveRequests and try to measure a load of 3000 users, the Tomcat server would try to open at least 3000 threads, one for each user session. The JVM used in this project only supports 1024 threads. This issue is described in the Appendix, Section C.6

6.5.4 JBoss application server

JBoss version 3.2.5 is chosen as application server since it is open source, well documented and easy to use. No modification of the server configuration is needed, other than deploying the *.jar files required by BankApp in correct JBoss folders.

A production-environment application server will typically be one from the major vendors, such as IBM's WebSphere, BEA WebLogic, or Oracle's integrated solutions. But for development and testing purposes JBoss is often preferred, since it is easy to deploy an application without any extra configuration files. The commercial application servers on the other hand often require the developer to create application server specific configuration files.

6.5.5 MySQL database server

MySQL is chosen as database server software because it is open source and well documented. MySQL is already used in the bank application example from EDB Bank & Finans. MySQL version 4.0.24 was built from source with no optimisations. To lighten the burden on the shared file system, the database files containing user data are copied to the local disk of the node where the database server is started.

MySQL offers several choices of database table types. MyISAM is the default table type and is fine for development purposes, and BankApp comes configured for this table type. Other table types are more effective and secure for transaction processing systems, but such issues are also beyond the scope of this project.

The database content are generated with a Java program that connects to the server. The Java program fills the database with random users. When the required amount of users are created, the database must be indexed explicitly.

The task of creating a database with a given number of users consists of many steps The steps are automated and documented in a script. This script creates data files for Grinder, so that the load generator knows which customers to log in.

6.5.6 Grinder load generator

The Grinder [21] is a Java load testing framework. Version 3.0-beta22 is used in all measurements. Grinder can be scripted to act as a human that sends requests with a browser. Test scripts are written in Jython, a Java implementation of Python. Jython combines the ease of writing scripts with Python running in a Java Virtual Machine. The result is that one can use both Java and Python libraries in a Jython script. See Appendix H for the Jython test script used in measurements.

6.5.7 Sysstat sampling tools

Sysstat is a system-level performance monitoring package. It contains various tools for sampling and collecting performance data. See [22] for more information. To instrument a node, SAR is started in the background. It then reports various information to a log file which can be analysed after the measurements.

Sysstat reads selected cumulative activity counters from the operating system, it does not actually instrument the system.

This means that when sampling the CPU utilisation you get the mean from the last interval. Therefore it is possible to get the mean utilisation by sampling in one long interval, instead of calculating the mean of many small sample intervals. But using such a method one looses important data for use in the debugging process. Bursts of any kind are more easily spotted when utilisation is sampled in small intervals.

7 Scaling objectives

Before assessing scalability of a system, the goals of the analysis must be set. We must determine what scaling functions we want to analyse. Then we must decide what scaling paths to investigate by considering the possible configurations for both hardware and software.

7.1 Operational context

Below we list a few possible scenarios that act as motivation for a scalability analysis. The list is not complementary, but consists of scenarios we find likely to be of interest to further research, as well as being relevant to real-life applications in the industry.

- Ground-up development of a new system and platform. The baseline system may not be available.
- Development of new applications to a specified platform. The baseline system is often available.
- Capacity management of an existing system, where several changes are expected: change in workload, server topology and platform or technology evolution. The baseline is available, and an upgraded system may be available.
- In research context, a laboratory experimenter may have access to the application and both a baseline and an upgraded system.

The focus in this project is the research context. The main reason is that we have the opportunity to measure the application not only on a reference baseline, but also on an upgraded system.

7.2 Software and hardware scope

A full and thorough scalability analysis would require amounts of work. Models must be made built to predict scalability of systems not yet built. The more detailed the model of a system is, the more parameters it needs. Obtaining parameters is time consuming since it requires specialised measurements, profiling or analytical techniques such as code walkthrough. A detailed model may result in a huge scalability exploration space, so one has to focus on chosen aspects. It is important to consider where to put the details, and where to simplify.

Choices must be made about what aspects of the system to study, and how detailed these studies should be. As mentioned in the introduction, the J2EE architecture is the primary focus.

Scope is reduced by focusing on higher levels of software:

- Java classes and J2EE software components.
- Java Virtual Machine and garbage collection.

Other parts of the system are left out of the study, if possible:

- Operating system inner workings (paging, system calls)
- Disk solutions (IDE/SCSI/RAID). We disregard the disk on Web and Application server, since they only use the disk when starting up the software and reads the configuration files. After that, everything runs in the memory. [1]
- Logging: The servers would not normally write log to internal disks, but rather send logging information to a logging node, or a cluster of logging nodes. Any logging functionality is disregarded in the analysis.
- Database inner working are also disregarded, since it is considered commercial "off-the-shelf" software and we leave it to the manufacturer to guarantee high performance and scalability. Database design is reserach field of its own, and we choose to deal only with the end-effects of the database[2]

The boundary of the system is defined to include the web, application and database server. The rest is outside the boundary. We assume that these "outside" components have sufficient capacity, and that they do not affect the resource usage on the servers. [3]

- Client architecture (browser software and hardware etc) [4]
- Wide Area Network architecture
- Firewalls and load balancers

7.3 Scaling

The three dimensions processing, connectivity and storage has individual scaling paths. We will focus on the processing dimension since it must be present in any scalability analysis. It is also the dimension where we have gained most experience.

Scaling the requirements

We want to scale the requirements with strict scaling, i.e. we use the same scale in all dimensions.

Scaling the platform

To meet a scaling of the requirements, we try to obtain uniform scaling of the platform. The available hardware constrains the possible scaling. Since we have access to slower nodes on the cluster in addition to the normal nodes, we are able to investigate upgrading of the system. We do not have the option of replicating the CPUs or the disks.

Replication of the servers was one possible scaling path that we considered, but that would require difficult configuration of the system and it was left out.

7.4 Scale invariants

The scale invariants enables us to compare systems of different sizes by eliminating certain aspects of the systems:

[1] The case study software is set up to run from a shared NFS disk storage, so the local disks are actually never used.

[2] We consider the end effects of database software and hardware. The database is regarded as a black box, where response time is the key metric. See Section Section 13.4 for details.

[3] If the outside components has sufficient capacity, they only impose a slight overhead in the total transaction response time. We assume that the outside components are not load dependent.

[4] We disregard client scale-up in the analysis, since a faster browser does not help the user think and decide faster.

- Work invariant: The workmix remains unchanged throughout the analysis.
- Congestion invariant: The congestion in the system is fixed with the operating point. The operating point is held for all scale factors of the system.

To ensure strict and uniform scaling, other invariants are used. The invariants are relationships that are kept constant. We classify the invariants after the hierarchic levels presented inSection 3.10.

Level I:

- #Users in database vs. # Scale factor [5]

Level II, Replication:

- #Web servers vs. #Application servers vs. #Database servers

Level II, Upgrade:

- Service rates for: Web server vs. Application server vs. Database server [6]

Level III Upgrade: Uniform scaling in a server

- CPU instruction rate vs. JVM heap size

Level III Upgrade: Uniform scaling across all servers

- CPU instruction rate for Web, Application and Database server

[5] Ideally, we would want to have one database setup for each value of load we want to measure, but this is not regarded feasible. We relax the invariant so that we only need one database size for each scale factor.

[6] This relationship is kept constant by using the same nodes for the three servers.

8 Workload specification

In this chapter we present an introduction to the workload used in the measurements and models. We idenfity the services offered to the client by the system, and then show how a user session workmix is constructed. Finally we show how to implement the workmix with the Grinder load generating tool.

We use the workload specification from Fagerlie-Landmark, see chapter 5 in [9] for a description of how they constructed the workload.

8.1 Top level operations

The system offers services to the client user, and these services are regarded as top-level operations. [1]

It is relatively straightforward to identify the top-level operations in a web application system, as the operations can be identified as the web pages requested by the user requests with a browser. HTTP forms enables the web requests to contain data from the user to the server.

The full list of the operations in BankApp is given below. Each operation represent a web page request:

- Initialize (main page)
- Login
- ViewPayments (get list)
- ViewPaymentDetails (view details)
- NewPayment
- ConfirmPayment
- PaymentReceipt
- Logout

BankApp has three more operations: "CreateAccount" ,"CreateCustomer" and "ChangeLocale". Since customers seldom create accounts or change locale, and most certainly never are allowed to create new user accounts, these operations are not used in the workmix.

8.2 Transaction

If a user wants to check a specific payment already registered in the bank, he typically has to request several web pages in order to complete such a task. A transaction is a sequence of logically connected operations.

For example, a "view payment"-transaction would consist of these operations:

[1] The services are commonly called *requests, or web requests,* but in SP terms they are called operations.

Initialize - >Login -> ViewPayments -> ViewPaymentDetails

If a user is already logged in, only the following steps would be performed:

ViewPayments -> ViewPaymentDetails

In the next chapter we will see that a transaction also can be viewed as the whole user session. All operations performed by a user, including login and logout, are considered to be one transaction.

8.3 Work-mix

To be able to compare measurement results, the mix of operations must be invariant. This means that all users will perform the same operations in the same order, where only service times and sleep times varies for each user.

The first step is to define the mix of operations performed by a typical user. Fagerlie-Landmark presents a graph of possible paths between the top-level operations. This graph illustrates for example that one first must log in before the "view payment" operation is available. A subset of the possible paths should be used as basis for defining the work-mix, expressing the typical user. The workmix must also reflect what one want to measure, i.e. the mix of read and write intensive operations.

Two transactions are chosen by Fargerlie-Landmark:

- View old payments (for example reviewing monthly automatic payments)
- Make new payments

A user performs each transaction three times. Below is an example of pseudo-code implementation of the work-mix:

```
Initialize
Login
do 3 times:
      ViewPayments
      ViewPaymentDetails
do 3 times:
      NewPayment
      ConfirmPayment
      PaymentReceipt
Logout
```

8.4 Concurrent user sessions and think time

The load is the intensity of the work put on the system. The intensity must be defined in terms of a unit. We choose *concurrent user sessions* as the unit. The load is a measure of how many concurrent user sessions there are "in" the system on average.

A user session lasts from the moment a user logs in and until he logs out. The session consist of requests and think time between the requests. Think time denotes the time spent by the user while he is thinking, reading or writing information before he is ready to issue the next request. Think times for a typical BankApp user are defined to be in the range of 5-120 seconds. See table 5-2 in [9]

for the original specification. The duration of a user session is determined by the total of all think times in the workmix: 515 seconds.

Operation	Think time before operation	Workmix	Total think time
1	10	1	10
2	10	1	10
3	15	3	45
4	10	3	30
5	10	3	30
6	120	3	360
7	5	3	15
8	15	1	15
Session total			515

Table 8.1: Think times of a typical Bankapp user

A user session means that the user is *in* the system, that he either thinks or accesses it. Even though the user is thinking and has no requests in the system, he will utilise some resources on the server even, like cached data or thread resources (if the HTTP connection is persistent). So a load of 1000 concurrent user sessions or "users", does not mean that 1000 users are accessing the system at once.

The arrival rate is then how many users logs in per time unit. If the system has reached equilibrium, the arrival rate is also the service rate of the system.

8.5 Load

For a load of 1000 concurrent user sessions, the user session throughput is 1 transaction / 515 seconds * 1000 = 1.9 tps.[2]

Measurements in Chapter 10 show that the server completes each top-level operation in tens of milliseconds, and the total measured response time for the workmix is a few hundred milliseconds. Clearly, most of the user session is think time.

The system will be measured in the range of what it can handle for a given workmix and configuration. Initial measurements show that the system can handle a load up to 1300 users on Clustis1 nodes, and 3400 on the Clustis2 nodes. With higher loads, the response time quickly increases to seconds.

8.6 Implementing the workload with Grinder

This section describes how the workload is implemented using the Grinder load generator.

The users are simulated using Grinder. Several nodes run one Grinder agent each which in turn starts a number of Grinder worker threads. Each thread emulates one user session, and when the user has performed all requests in the workmix and logs out, the thread will load a new random user and start over again. Therefore there will always be approximately X user sessions in the system at any time.

[2]If our perspective was separate web requests instead of whole sessions, the throughput would be 9 / 515s * 1000 = 0.017 tps (since there are 9 page requests in the workmix). The top-level operations have different service demand requirements, and are thus not directly comparable.

1 Node -> 1 Grinder agent -> X Grinder threads

1 Grinder thread -> Y user sessions = [Session1, Session2, ... SessionY]

1 User session -> Z requests = [Request1, sleep, Request2, sleep, ... requestZ]

Typically a Grinder agent is started on 5 separate nodes. Each agent starts up to 300 threads on the baseline system. The total load on the system in that case is N = 5 * 300 = 1500 concurrent user sessions. For the upgraded system, the load is measured up to 3600 sessions.

The Grinder threads are configured to start with a random delay, and this delay is chosen by a random uniform distribution between 0 and 515 seconds, which is the average time for a user session. After 515 seconds, all threads are active and the load is considered constant. Even though the load is constant, it must be investigated when the system is considered in steady state and ready for measurements. Grinder is configured by modifying the variable "initialSleepTime=515". This specifies the maximum time before all threads are started. See Figure 8.1 for an illustration of how the workload is implemented on one node.

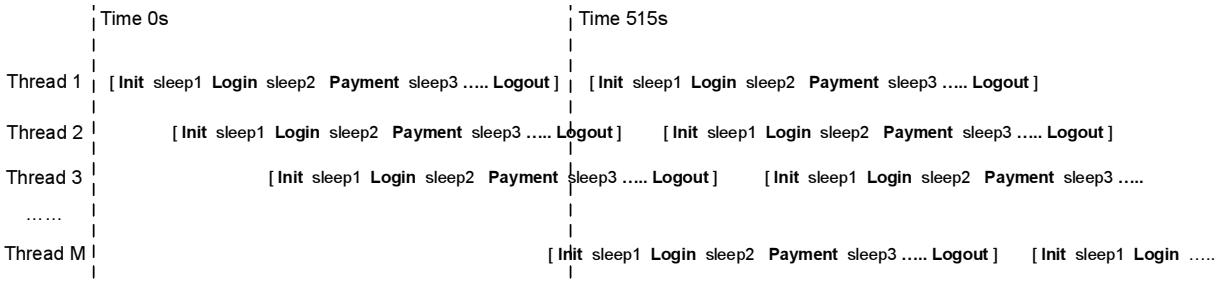


Figure 8.1: Workload implemented as Grinder threads on one node

The Grinder nodes are started one by one in 2 second intervals. The first Grinder node is then started at time t=0. On average, all threads on each node has started after 515 seconds. The last node will have started 2*4 seconds later, so all nodes are running full load onaverage at T=525seconds.

8.6.1 Grinder think times

Think time between requests are specified in the Grinder scripts as constants. See the next section for an example of a partial Grinder script. These constants are average think times taken from the work-mix definition in Table 8.1. To randomise the think time, Grinder is configured with the variable "sleepTimeVariation=0.4". The Grinder documentation states that this variable specifies the fractional range which nearly all the times will lie in. "Nearly all" is defined as 99.75% of all times. For a value of 0.4, 99.75% of the sleep times will vary between 600 and 1400 milliseconds.

Grinder uses a normal distribution to vary the actual length of the sleep() method. Ideally one would want an exponential distribution which has memoryless properties[3], but Grinder only supports the normal distribution at this moment. But other factors are believed to make the design more memoryless. The Grinder agents are started up with approximately one minute delay between them. Each agent will start many threads that compete for resources, and then the Linux thread scheduler will start the threads in a non-deterministic fashion. This is shown by the fact that there is inconsistency of when a thread is scheduled to wake up, and when the thread actually starts(seen in the Grinder logs). It is assumed that these unpredictable delays removes the need for a memoryless distribution.

[3]Memoryless properties means that the each new random value is completely independent of the last value. Using the exponential distribution as inter arrival times for customers means that one cannot decide when the next customer arrives by analysising the arrival of all customers that day.

8.6.2 Grinder script

This is a part of the load generating script, showing what requests are sent to the web-server. User think time is denoted by *grinder.sleep*(). The last line *self.data.loadNewCustomer()* instructs Grinder to load a new (random) user. The Grinder thread will then run the script again, emulating a new user session.

Loading random users is done to avoid caching effects on the database server. The web and application server do not cache the users. In any way, it is assumed that real-life users of an Internet banking application do not log in more than once a day or a week.

Listing 8.1: A part of the Grinder load script

```
grinder.sleep(10000)
test1.GET("http://node02/BankApp/index.jsp")
grinder.sleep(10000)
test2.POST('http://node02/BankApp/transigo/login', self.data.login())
for i in range(3):
    grinder.sleep(15000)
    test3.GET('http://node02/BankApp/transigo/payments')
    grinder.sleep(10000)
    test4.POST('http://node02/BankApp/transigo/paymentdetails', self.data.getPayment() )
for i in range(3):
    grinder.sleep(10000)
    test5.GET('http://node02/BankApp/transigo/newPayment')
    grinder.sleep(120000)
    test6.POST('http://node02/BankApp/transigo/confirmPayment', self.data.getPaymentConfirm())
    grinder.sleep(5000)
    test7.POST('http://node02/BankApp/transigo/paymentReceipt'
grinder.sleep(15000)
test8.GET('http://node02/BankApp/transigo/main')

self.data.loadNewCustomer()
```

8.6.3 Application state

The database represents application state, where user data are stored on persistent storage. The user base varies between 50 000 and 600 000 users in the database. There are 3 accounts and 11 payments per user. The largest database of 600K users contains therefore 1.8 million accounts and 6.6 million payments, and the database files requires over 1 gigabyte of disk storage.

To avoid unecessary load on the network, the database files are copied to the local disk where the database server software runs. The files are copied for each new experiment to guarantee that the measurements has equal starting conditions.

8.6.4 Users on client side

Some user info must be present for the Grinder nodes, so they are able to interact with the system. The information needed are both customer ID (account number) for logging in a user, and payment IDs so that the scripts can request info of those payments. This user info is extracted from the database and written to a file. This file is read by the Grinder scripts, and must be distributed to each Grinder agent. Grinder picks random users from this file.

Each line represents one user, where the first field is the account number (customer ID) and the 11 payment IDs for that user:

Listing 8.2: Users on client side

```
[Account number];[payment id 1];[payment id 2];[payment id 3].....[payment id 11]

10000003538;14810083082901;15147607134164;13820660869910;.....
32425323443;13245532892421;11694256328123;12567754332219;.....
10023123244;16321205932483;35322826754323;10023251263253;.....
```

9 Measurement toolbox

In this chapter we describe the work performed to build a "framework" of scripts and programs to ease the measurement burden, but also to produce more reliable measurements. Having one centralised configuration file for a measurement setup reduces the probability of measuring a misconfigured system.

The scripts are not intended as a complete program or framework, but they are rather intended to provide peers with a script *toolbox*. The toolbox contains useful UNIX knowledge, Bash and Perl scripting, how to run various programs and how to use the Clustis2 cluster.

First we describe incentive for building a toolbox, and then the measurement steps are described. The toolbox features and usage is then presented. Finally we evaluate the toolbox.

9.1 Why build a toolbox script library

Configuring and testing the servers and software between each experiment is time consuming, so it is desirable to automate as much as possible. Professional load generator software packages are one possible solution, except that they are usually very expensive. These general tools often require lots of conficuration and experience to be used correctly and efficiently.

What we achieve by scripting and automation:

- **Easy configuration**: All important variables are in one central configuration file, making it easy to check the setup.
- **Less error prone**: With one configuration file it is easier to remember to change important settings when switching between different experiment setups.
- **Repeatability**: It is easier to repeat a test when all configuration files are saved after an experiment. If anything went wrong it is easier to check the logs.
- **Documenting**: The scripts serves as documents on how all procedures actually are done in practice. There is no need to learn everything from scratch when using the scripts.
- **Time-saving**: laborious tasks are automated as much as possible. For example the procedure to generate databases with different number of users consists of many steps and it is easy to miss something when manually performing the procedure.

9.2 Measurement tasks

Measuring a system requires a lot of setup, configuration and testing. We often need to perform many measurements with different configurations, and we need to be able to easily reproduce the measurements.

We measure the system by following this recipe:

- Edit configuration files to reflect a new test, both server and client settings.

- Start servers one at a time by waiting until they are initialised before starting the next. This makes debugging easier.
- Start measurement tools on servers (e.g sysstat tools)
- Start clients and run load scripts. Both system warm-up scripts and the main load scripts.
- Collect logs from servers. Web, App and Database-server.
- Collect measurement data.
- Examine logs to check for errors or unusual values.
- Analyse measurement data to get performance indices.
- Plot indices on graphs to check if values are as expected.
- -> If not, check logs again to see if anything interfered with the measurements.

9.3 Cluster resource limitations

The toolbox is developed for use on a cluster of computer nodes like Clustis2 [18]. One challenge is the lack of dedicated computers for measurement purposes. The computer nodes available to us are on a cluster called Clustis2, which is a shared resource with many users. To distribute access between the users, Clustis2 has a batch scheduling system called OpenPBS. When the required number of nodes are available for a job in the queue, the job starts automatically on the assigned nodes.

The list of assigned nodes will most likely change for each job submitted. Since software components need to know where to send requests to each other, the configuration files must be updated for each measurement with the current node addresses. Needless to say, this complicates the scripting task and must be automated. Even though the OpenPBS batch scheduling system dedicates nodes to a job, anyone can log in to any nodes and start processes if they want. Even if users behave accidents can happen, so one must watch out for interfering processes.

Special care must be taken when designing the toolbox, since the cluster is a shared resource where several users can use it simultaneously. Since the measurement process is automated, we must make sure that we do not interfere with other users:

- Make sure that the correct nodes are used, not interfering with other users.
- Ensure that all processes on all nodes are killed after the experiments
- Clean up after the experiments, especially if large log files are produced. One solution is to automatically compress the files and archive them with the measurement files.
- Use fail-checking in the scripts: When issuing a copy or delete on a path given as environment variable, make sure that the environment variable exist. [1]

9.4 Overview

The toolbox is primarily built as bash shell scripts, but also some perl scripts were used for analysing data. Initially some of the analysing scripts were written in Bash, but the scripts executed too slow and were rewritten in Perl.

The main idea is to have one central configuration file and one run-file that describes how to perform the experiment. The configuration file may be used by all toolbox scripts. The run-file (experiment.sh) makes preparations, starts software, run the tests and finally prepare a report of errors and results, before it archives all experiment files into one folder.

[1] E.g: “rm -rf $GRINDER_HOME/logdir. If the variable does not exist, this command is evaluated to “rm -rf /logdir”

9.5 Structure of the toolbox:

The toolbox consist of two folders "Measurements" and "scalability"[2]. All software binaries are placed in "scalability", while configuration and log files are put in "Measurements". The most important parts are the "Setup" folders where all experiment related files are. The "Scripts" folder contains all scripts in the toolbox. To be able to run the scripts from anywhere in the file system, the Scripts folder must be added to the PATH environment variable. It can be added permanently by putting it in the ".bashrc" or ".bash_profile" files in the home directory:

```
export PATH=$PATH:$HOME/Measurements/Scripts
```

The listing gives and overview of the toolbox directory structure:

Listing 9.1: Toolbox directory structure

```
/home/geirbo/
  .reserved_nodes            List of nodes reserved by PBS scheduling system
  .current_run_path          Path to current experiment is saved here

  Measurements/
     Scripts/                Scripts for both automatic and manual operation
     Servers/                Server configurations and log-files, separated from binaries
        apache/              Apache load balancer configurations
        tomcat/              Tomcat configurations
        mysql/               Mysql databases (different sized)
     Setups/                 Main work folder, storing different setups
        baseline/            Baseline configuration
           config_templates  Configuration files with variables
           configuration.sh  Main configuration file
           experiment.sh     Performing the experiment
           experiment.log    Log-file from experiment
           node_mapping.sh   The role of each assigned node (web,app,grinder..)
           run-experiment.sh Start the experiment
           gc-jboss.log      Garbage collector output from JBoss
           gc-tomcat.log     Garbage collector output from Tomcat
           grinder/          Grinder scripts are placed here
           grinder_logs/     Grinder logs and results
           log_archive/      Experiment files are archived here in separate folders
           statistics/       Sysstat statistics files (sar)
        upgrade/             Upgrade configuration
        setupX/              Whatever setup you want (4web1app, upgrade_integrated..)
  scalability/               Software binaries (tomcat,jboss,grinder..)
```

Two important files are found under the HOME directory:

.reserved_nodes This file stores a list of nodes reserved for the job. The scripts use this list to assign nodes to the experiment.

.current-run-path This file describes the path to which setup is running at the moment. Scripts use this file to find the correct setup. One neat feature is that you can run most scripts from anywhere in the ClustIS file-system, and this file will point the scripts to the correct setup.

9.6 List of toolbox script files

This is a list of the files in the *Scripts* directory. They are the backbone of the toolbox, used by the experiment setup files *configuration.sh* and *experiment.sh*, but some are also used manually by the experimenter to output info about before an experiment has completed.

[2]The folder name "scalability" is a remnant from when the software was put under /opt/scalability. The software is now put in the home-directory, and really should be renamed to "software".

Listing 9.2: List of toolbox script files

```
abort-experiment.sh                Abort the experiment hard
analyse-gc-log.pl                  Outputs analysis of logs
analyse-grinder-logs.pl                   - " -
analyse-mysql-logs.pl                     - " -
analyse-sar-log-for-parameters.pl         - " -
analyse-sar-log.pl                        - " -
check-logs.sh                      Check all logs for error messages
check-node-status.sh               Checks the status of all nodes (logged in users etc)
check-reserved-status.sh           Check what nodes are reserved
cleanup-tmp-files-on-nodes.sh      Clean up temporary files on local storage on nodes
collect-experiment-logs.sh         Gather all experiment logs, compress and arhive the files
create-arrival-graph.sh            Produces a graphs underway or after an experiment
create-gc-graph.sh                        - " -
create-graphs.sh                          - " -
create-paging-graph.sh                    - " -
create-utilisation-graph.sh               - " -
format-gc-data-for-gnuplot.pl      Formatting log file data to use in graph scripts
format-paging-data-for-gnuplot.pl         - " -
get-current-run-path.sh            Get the run path for current experiment setup
get-nodelist.sh                    Get list of assigned nodes for the last OpenPBS job
get-time-difference.pl             Computes the time difference in seconds between two dates
kill-processes.sh                  Script used to kill experiment-relevant processes on a node
prepare-4web1app.sh                Preparing configuration files for 4web to 1 setup
prepare-bankapp.sh                 Alter configuration files and compiles BankApp and deploys it
prepare-node-mapping.sh            Assign nodes to roles: 'compute-0-1' to application server
prepare-templates.sh               Alters template/skeleton files with configuration variables
reserve-nodes-interactivejob.sh    Reserve an interactive job. Run "get-nodelist.sh" afterwards
start-grinder-on-nodes.sh          Log in to nodes and run "start-grinder.sh"
start-apache.sh                    Starting software on nodes. Must be logged in to node first
start-browser.sh                          - " -
start-grinder-init.sh                     - " -
start-grinder.sh                          - " -
start-jboss.sh                            - " -
start-loadbalancer.sh                     - " -
start-mysql.sh                            - " -
start-statistics.sh                       - " -
start-tomcat.sh                           - " -
stop-experiment.sh                 Stop the experiment.Bit different than abort-experiment.sh
stop-servers.sh                    How to stops the servers nice. (old script)
view-apache-logs.sh                View the logs underway in an experiment
view-grinder-logs.sh                      - " -
view-jboss-logs.sh                        - " -
view-mysql-logs.sh                        - " -
view-tomcat-logs.sh                       - " -
view-statistics-logs.sh                   - " -
view-connection-pool-stat.sh       Outputs statistics on number of active connections in pool
view-parameterising-summary.sh     Outputs summary for stepwise parameter measurements
view-summary.sh                    Outputs experiment summary, using many scripts
view-thread-count.sh               Outputs thread count on the server nodes
```

9.7 Experiment overview

The experiment.sh file describes all actions to be performed in an experiment. It prepares configuration files and synchronise the software.

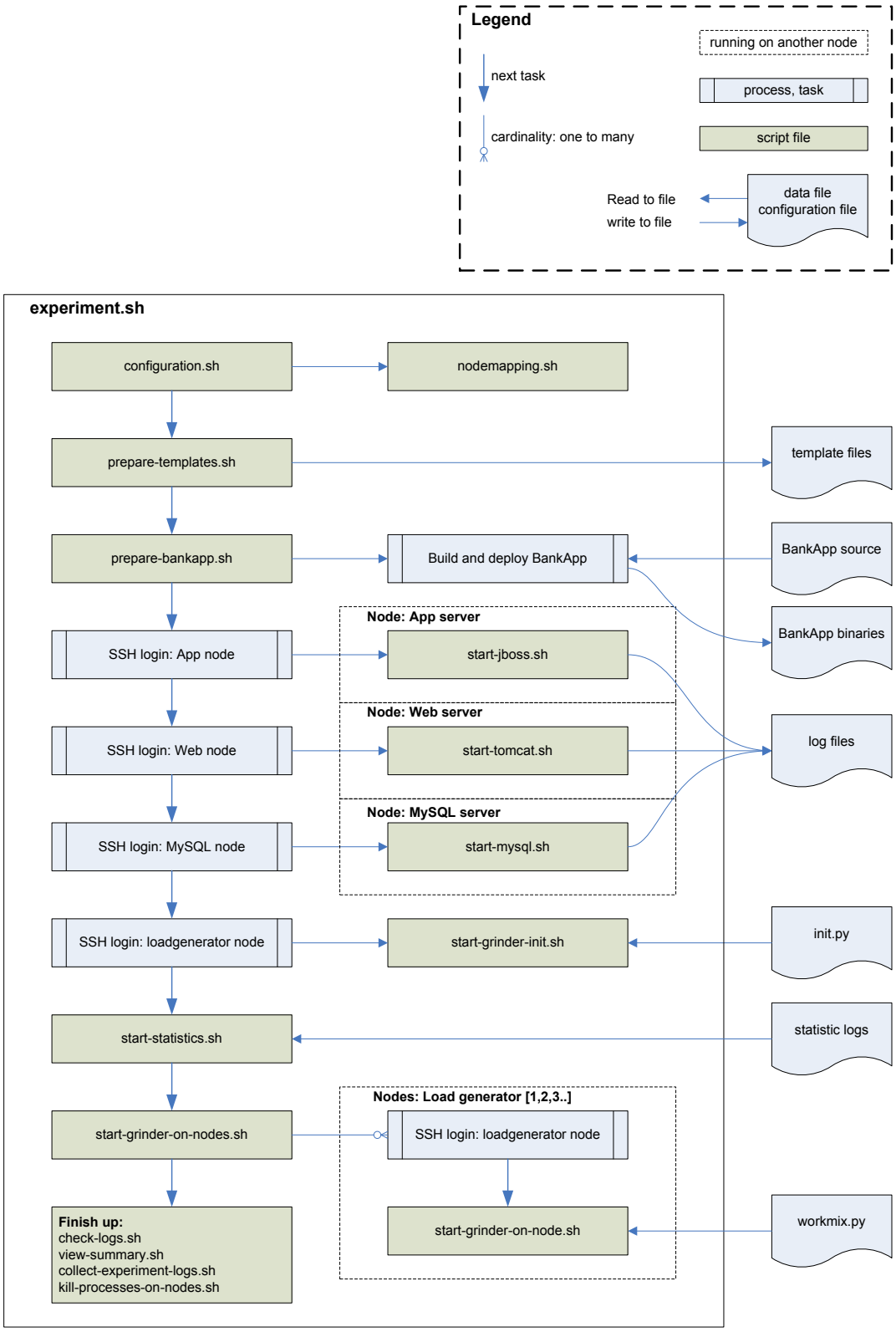


Figure 9.1: Experiment.sh

9.8 How to run a new experiment

This is a short guide on how to setup and run a new experiment.

1. Copy a setup folder that best matches the new experiment.
2. If an an old experiment setup is copied, and variables or other things have changed since then, it may be necessary to run the command "diff" on some of the configuration files. Compare with the newest working configuration files from another setup to determine what has been changed.
3. Edit "configuration.sh" and "experiment.sh" to reflect the new experiment.
4. Start with"run-experiment.sh".
5. To view experiment output, run "cat experiment.log". Check for errors with "check-logs.sh". View a summary of the results with "view-summary.sh"

9.9 Toolbox in action

Logs should be analysed as automatically as possible. When a new error is found in the logs, a rule should be put in the "check-logs.sh" file. Counting number of errors, and showing a few lines the error messages greatly helps in the problem solving task. This is an example output from an experiment:

Listing 9.3: Output from check-logs.sh

```
========================================
Unique customers (from grinder logs)
========================================
Total  customers: 35778
Unique customers: 29471

========================================
Grinder errors
========================================
Connection refused:  385
error_compute-0-20.local-0.log:8/31/06 7:23:02 PM (thread 106 run 10): Aborted run, script threw
            class java.rmi.ConnectException: Connection refused to host: 10.255.255.242; nested
            exception is:
error_compute-0-20.local-0.log: java.net.ConnectException: Connection refused
General errors:      728
error_compute-0-20.local-0.log:8/31/06 7:23:02 PM (thread 615 run 10): Aborted run, script threw
            class java.rmi.ConnectIOException: error during JRMP connection establishment;
            nested exception is:
error_compute-0-20.local-0.log:java.rmi.ConnectIOException: error during JRMP connection
            establishment; nested exception is:
IndexError:          0
SQL Duplicate entry: 593
Transaction errors:  701      (http response not "200 OK")

========================================
Tomcat errors (Grinder logs)
========================================
server:              11004
error_compute-0-19.local-0.log: at org.jboss.mx.server.ReflectedDispatcher.dispatch(
            ReflectedDispatcher.java:60)
error_compute-0-19.local-0.log: at org.jboss.mx.server.Invocation.dispatch(Invocation.java:61)
internal:            1186
error_compute-0-19.local-0.log: at org.jboss.ejb.StatelessSessionContainer.internalInvoke(
            StatelessSessionContainer.java:331)
```

```
error_compute-0-19.local-0.log: at org.jboss.ejb.StatelessSessionContainer.internalInvoke(
            StatelessSessionContainer.java:331)

=========================================
Tomcat errors (Tomcat logs)
=========================================
Already started?
memory?
exceptions:         0
threadpool:         0
severe:             0


=========================================
JBoss errors (JBoss logs)
=========================================
already started?
shutdown?
server.log:2006-08-31 19:23:02,181 DEBUG [org.jboss.system.server.Server] Shutdown complete
memory?
Warn:                 0
Error:                1
```

The "view-summary.sh" script runs several scripts to analyse the logs. The script analyses the logs in a given interval, for example 3000 seconds in the experiment, with a duration of 2000 seconds.

The scripts then calculate the mean CPU utilisation and how much CPU utilisation was used for garbage collection. More detailed info is also given about the average and maximum heap size. The load generator logs are analysed to get the mean response time for each request and the number of requests that were issued. The throughput is calculated based on the interval length and the number of "logout" requests(request 8)

Listing 9.4: Output from view-summary.sh

```
================================================
>>>N=2500<<<  (5 nodes * 1 workers * 500 threads)
           Heapsize: 700 MB, Database: 50K
           Analysis start:3000 sec, interval: 2000 sec
           Date: Fri Aug 18 00:22:14 CEST 2006
           Folder: 20060818-0022
================================================

Tomcat:
  Utilisation: 65.23%  ( 400 samples, sum:26091.4)
JBoss:
  Utilisation: 18.28%  ( 401 samples, sum:7330.1)
DB:
  Utilisation:  5.99%  ( 401 samples, sum:2400.1)

Tomcat:
  GC CPU usage: 1.51%  (1071 samples [full:33, tenured:0] sum:30.15 sec, sample time:1999 sec,
              heap[avg:146MB,max:158MB]
JBoss:
  GC CPU usage: 0.30%  ( 404 samples [full:34, tenured:0] sum: 6.05 sec, sample time:1999 sec,
              heap[avg: 78MB,max: 79MB]

Mean response times:
Test  Mean(ms)       #samples     Ratio  Workmix      Sum   Std.dev
  1        7.7          9688     1.000        1    74765     33.18
  2       74.3          9687     1.000        1   719387    100.92
  3       46.7         29092     3.003        3  1359435     77.17
  4       46.6         29094     3.003        3  1354393     74.33
  5        8.4         29077     3.002        3   244275     34.31
  6        9.1         29073     3.001        3   264529     32.91
  7       38.4         29068     3.001        3  1116277     69.50
  8        6.9          9691     1.000        1    67207     29.81

Total response time:    536 ms  (workmix multiplied with mean response times)
Throughput:           4.846 tps (users leaving system)
```

The view-summary.sh script was modified for use in single user stepwise measurements. Were we show the output for the resource usage of the Init + Login + NewPayment transaction. The goal of this script was to a more automated process of obtaining parameters.

Listing 9.5: Output from view-parameterising-summary.sh

```
============================================================
Number of invocations: 60000
Heapsize: 700 MB, Database: 50K
Date: Mon Jul  3 16:01:52 CEST 2006
============================================================

Grinder start time:
7/3/06 3:24:54 PM (process compute-0-17.local-0): starting threads
Grinder stop time:
7/3/06 4:01:30 PM (process compute-0-17.local-0): finished

Experiment duration: 2196 s
Throughput:          27.32 tps

Tomcat:
Average utilisation: 64.17%    (samples:440, sum:28232)
Transaction Sdemand: 23.49ms

Jboss:
Average utilisation: 17.70%    (samples:439, sum: 7768)
Transaction Sdemand:  6.48ms

DB:
Average utilisation:  1.24%    (samples:439, sum:  544)
Transaction Sdemand:  0.45ms

             Transactions Errors        Mean      Mean      Response      Response
                                        Transact  response  bytes per     Errors
                                        Time (ms) length    second

Test 1   60000   0    3.64   9758.00   ?         0         "request main page"
Test 2   60000   0    27.66  9860.00   ?         0         "login"
Test 3   0       0    ?      ?         ?         0         "payments"
Test 4   0       0    ?      ?         ?         0         "payment details"
Test 5   60000   0    4.55   12560.00  ?         0         "new payment"
Test 6   0       0    ?      ?         ?         0         "confirm payment"
Test 7   0       0    ?      ?         ?         0         "payment receipt"
Test 8   0       0    ?      ?         ?         0         "logout"
```

9.10 Toolbox scope and extensibility

As mentioned before, the toolbox is not a general software package intended to solve all measurement needs. It is rather application domain specific and must be tailored for specific domains. We list the scope of the toolbox and possible extensions:

- **Platform:** The toolbox is built to work on Linux/Unix platforms. The toolbox is built to work in a heterogenous environment of nodes with shared storage available from any nodes, including the master node. With some configuration the toolbox should work with alternative job scheduling software. The toolbox only needs a list of available nodes from the scheduling software produces.
- **Possible platform:** It should be possible to run the scripts on Windows in an Unix environment, such as Cygwin[3]. This is not tested, and we only outline a few possible solutions and challenges. The toolbox makes extensive use of SSH login to the nodes to perform tasks, and

[3]See `http://www.cygwin.com/`

this requires a running SSH server on Windows nodes. The scripts also uses a lot of piping, and there are(or have been) issues with piping in Cygwin.

- **Software:** It is built around Grinder, Tomcat, JBoss and MySQL, but it should be easy to replace the software. It is just a matter of altering the configuration file to reflect new variables, modify the existing startup-scripts to the new software, and finally edit the experiment file to run the new software.
- **Hierarchic considerations:** The toolbox makes it possible to use several web servers connected to one application server, but the toolbox is not configured for replicating the application server. Such functionality can be added when needed, but it will complicate the scripts a bit. It is also configured to use one database server, but this should not impose a problem unless the server must deliver large, binary files.

10 Pilot measurements

This chapter presents how the BankApp application is measured. Pilot measurements are shown to get a feel for the level of modelling detail needed, and to prepare for the parameter investigation in the next chapter. We also briefly outline the various measurement designs.

The pilot measurements establish confidence in the measurements, indicating how the system acts under load, and how long it takes before the system reaches steady-state. The measurement results presented in this chapter acts mainly as a visualisation of the live running system, how it behaves under increasing loads and in what ranges the system can be measured.

Pilot measurements may reveal problems under high loads, giving the experimenter a chance to reconfigure the experiment design in an early stage, instead of having to repeat measurements after discovering the problems afterwards.

10.1 Test harness overview

To avoid confusion about the measurement methods, the types will be briefly explained here. There are two main groups of measurements: single user and multiple user measurements. The key factor that separates them is concurrency at the top level of the system, i.e. how many users that may access simultaneously.

Single user measurements There is only one user accessing the system at any time, and think time is reduced to zero. This type of measurements are mostly performed for parameterising purposes, when the goal is to obtain the resource usage without contention.

Stepwise single user measurements These are single user measurements where we want to obtain the resource usage for each operation in the workmix. If some operations depend on each other we measure parts of the user session in steps. One example of dependence is when the user has to login before he can perform any other tasks. The procedure is described inSection 13.2.

Multiple user measurements Several users access the system concurrently. Users access the system according to the specified workmix and mean think times. The goal of these measurements is to capture the resource usage when there is contention in the system.

For a general and more formal description of the possible test designs, see [6].

10.1.1 Single-user measurements

The goal of single user experiments is to capture static properties of the system. The results are service demands for each hardware device. The measurements are mostly used in these cases:

- **Calibration**: Single user stepwise measurements are used to calibrate the output from the static model.

- **Parameterisation**: For simple SP models, such as using only one SP software component for each node, the single user measurements can be used to parameterise the dynamic model directly, omitting the SP model. This is the case in this project.
- **Scale factor benchmarking**: To quantify the scale factor of a system, single user measurements is one option. Measure the time it takes to complete for example 10000 user sessions with the work-mix definition. Compare this time with measured time for the baseline system to obtain the scale factor in the processing dimension.

These are typical properties that may be changed in these measurements:

- New random users for each user session, or let one user log in over and over again to investigate caching effects.
- Heap size, database size and connection pool.

10.1.2 Multiple user measurements

In multiple user measurements concurrency is introduced, and the system is measured under "real" load. The measurements can be used for these tasks:

- **Validation:** The models are validated using multiple user measurements. The validation show if the models correctly predicts the system's capacity at some level of contention.
- **Load dependent effects**: These effects can be quantified by measuring the service demand as the load increases on the system under test.
- **Non-linearity search:** When searching for non-linearities, the baseline and upgraded system can be measured under high load and the results are compared to each other.

10.2 Performance indices

When measuring a system the goal is to extract data that indicate the performance for a given workload. The main performance indices obtained in our measurements are:

- Utilisation of CPU and disk (measured by Sysstat)
- Mean response times and throughput for requests (measured by Grinder)
- I/O activity, paging activity per time unit (measured by Sysstat)
- Garbage collector CPU utilisation (JVM garbage collector logs)

10.3 Pilot measurements

Before doing any parameter, baseline and validation measurements, it is recommended to get an idea of how the system behaves under different loads, giving some early indications of how the measurements should be configured. These pilot measurements also acts as an error-check. A lot of errors and problems were detected in the measurement work, see Appendix C. Pilot measurements can answer some of these questions:

- Does the load-generator work properly? Modify the load scripts to correct any errors.

- What is the utilisation level on the different nodes? Can all nodes reach full utilisation? If the web server is highly utilised and the application server is only 30% utilised, then one may need to use several web servers per application server to achieve a higher level of utilisation on the app server..
- How does the garbage collector behave? The heap size is adjusted to see how the system responds, and try to find an optimal size. It is important to remember that if we want to measure on a system that is strictly scaled by a factor of, say 2, there must be enough memory to allow a doubling of heap size.
- Is there any paging activity? The approach is to stress-test the system, preferably on the upgraded system, and see if the activity increases proportional from the baseline.
- What operating point to use? Check the CPU utilisation leves. Mean transaction response time for the requests can also give an indication. If we study a graph showing transaction times per number of concurrent user sessions, see Figure 10.4, we can choose the operating point before the service demands exceed some threshold.

By plotting a graph of the measurement data, it is easy to locate results that deviate. A higher utilisation or service demand than expected often indicates interference from other processes stealing CPU time. If the utilisation is lower than expected, several scenarios are possible:

- HTTP requests did not reach its target because of invalid IP addresses (errors in the scripts)
- The request queue in the HTTP server is too small.
- Requests were rejected if the web or app server could not handle the load for some reason.
- The heap memory could be full

For each measurement, examine several logs to check whether the experiments are valid.

- *Database logs:* The first thing to check is whether logs are empty. Then we know that something is wrong, if no requests went through to the database server.
- *Load generator logs:* Number of unique users, response time for each request. Very high response times often indicate limited thread resources.
- *Application and web-server logs:* Did the servers start, and did they run without any errors? Web server logs must be checked to see the maximum number of threads were in use.

10.4 Pilot measurements results

These measurements show how the utilisation increases as the load increases. The baseline results are performed on ClustIS(1), and the upgrade measurements are performed on Clustis2.

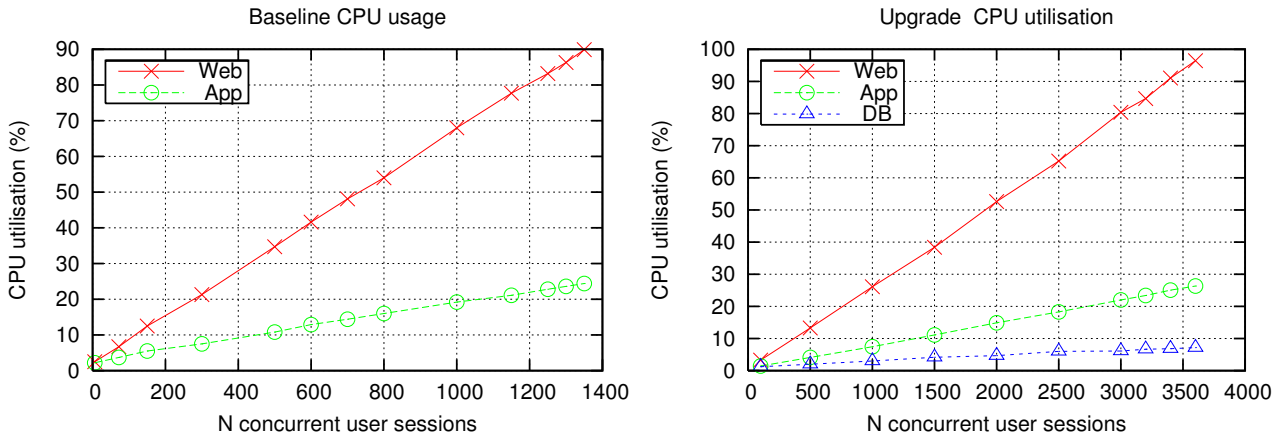


Figure 10.1: CPU utilisation for baseline and upgrade

Figure Figure 10.2 compares various measurement results to eachother. The measurements are performed on Clustis1 and Clustis2 as noted in the legend. The heap size was 400MB for all measurements except for one. The graph show that the baseline validation measurement on Clustis2 is a few percent lower than the Clustis1 measurements.

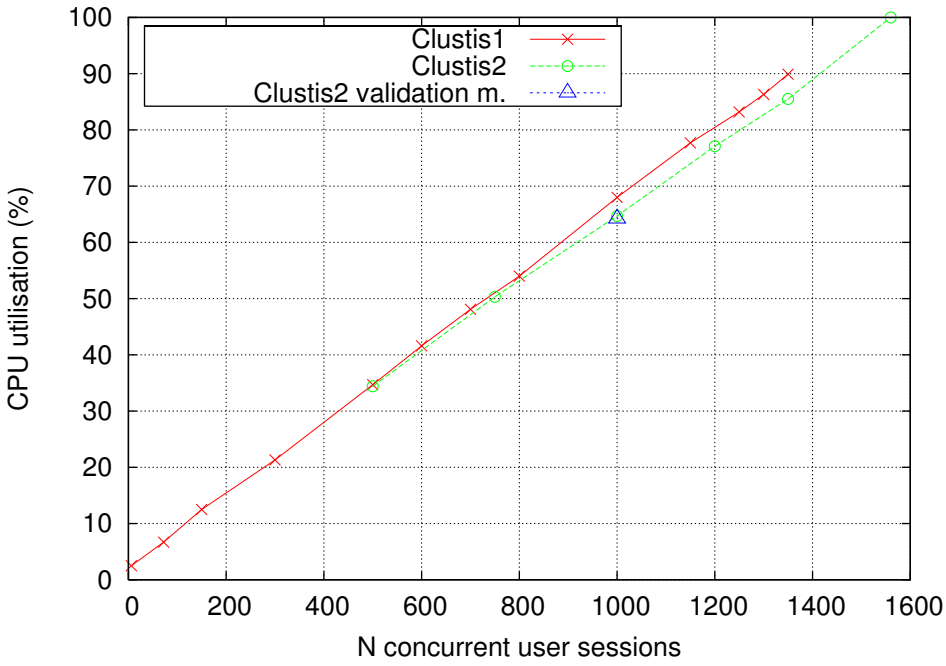


Figure 10.2: Baseline Web server: Comparing various measurement utlilisation results

Garbage collector usage is obtained by analysing log files from the JVM. The procedure is described in section 13.3.

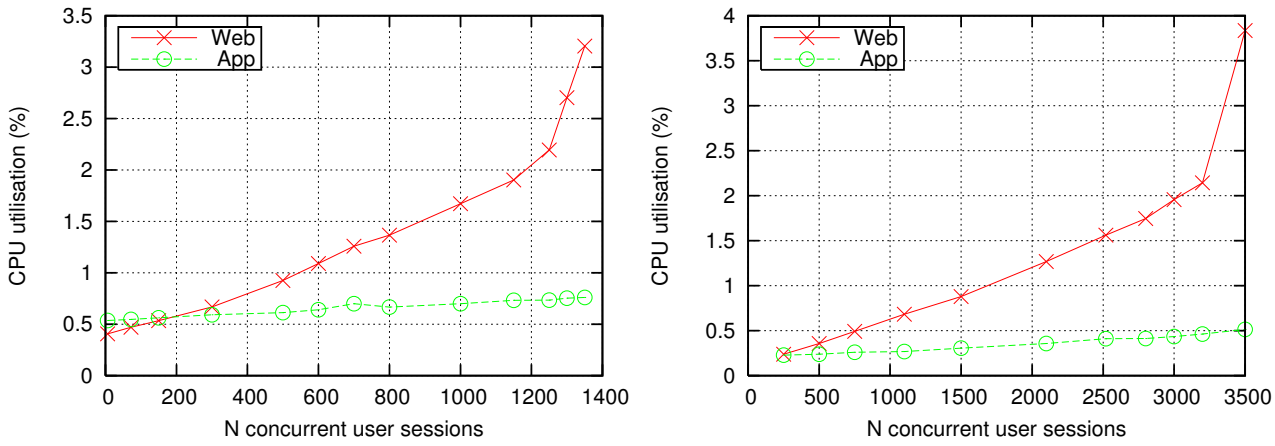


Figure 10.3: Garbage collector utilisation for baseline and upgrade

Grinder reports the transaction response times to log files. This is the elapsed time from a Grinder thread sends a request till it receives server response.

Three request types are chosen out of the eight that are measured in the work-mix.

- *Init* is chosen for its minimal service demand, it tests the minimum response time for a simple static HTTP request.
- *Login* is chosen since it always requires a database lookup, and starts a user session.
- Finally, *payment* is chosen for it's high service demand. This is the request that requires most processing on the application server.

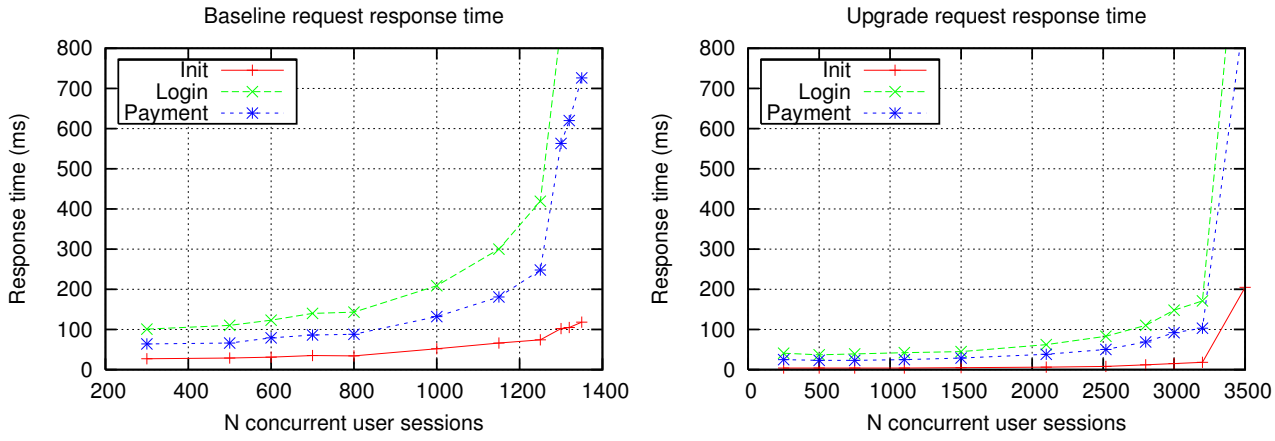


Figure 10.4: Request response times for baseline and upgrade

Finally, Figure 10.5 show the mean response time pr. user session and the throughput in terms of completed user sessions per second.

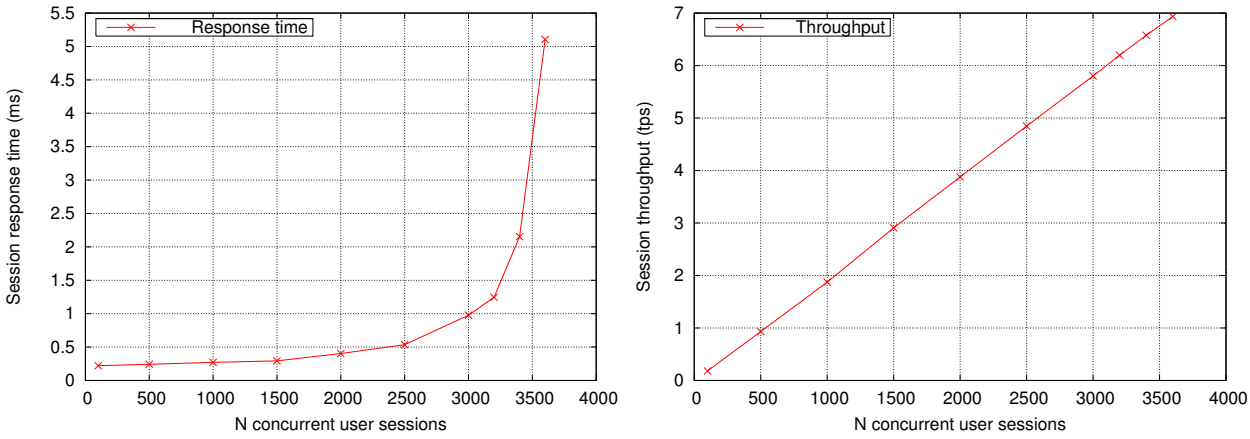


Figure 10.5: Upgraded system: Session response time (left) and Session throughput (right)

10.5 Experiment variation

Two experiments were performed with the same parameters to check the variability between measurements. The graphs show the utilisation on the web server. Upgrade nodes were measured with a load of N=550 from one Grinder node. This means that 550 Grinder threads were running on one node. Heap size was 150MB.

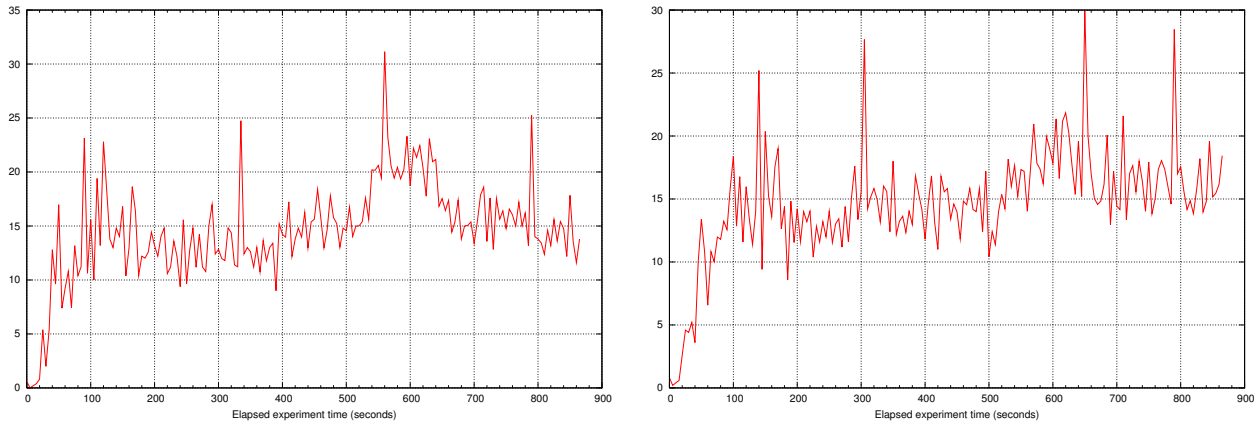


Figure 10.6: CPU utilisation on web server. Experiment 1 (left) and experiment 2 (right)

These graphs show that there are some variation between the experiments. The experiments show three clear bursts after 200 seconds. One of the bursts happens at 790 seconds for both experiments, but the other bursts happen at different times. Considering that the load on the servers will be the combined load from five such load generators, and the fact that we start capturing data after some warm-up, this should provide enough variation between experiments.

10.6 Steady state: arrival rate

The upgraded nodes were measured with 5 load generators running a total load of N=2500 and heap size of 700MB. The graphs in Figure 10.7 shows how many customers that arrived for various

intervals. The graph to the left shows the data partitioned in 60 second intervals and the graph in the middle in 20 second intervals. The graph to the left shows the data partitioned in 1 second intervals, in a shorter interval of <9900,10000> seconds. We can see that the load is not bursty.

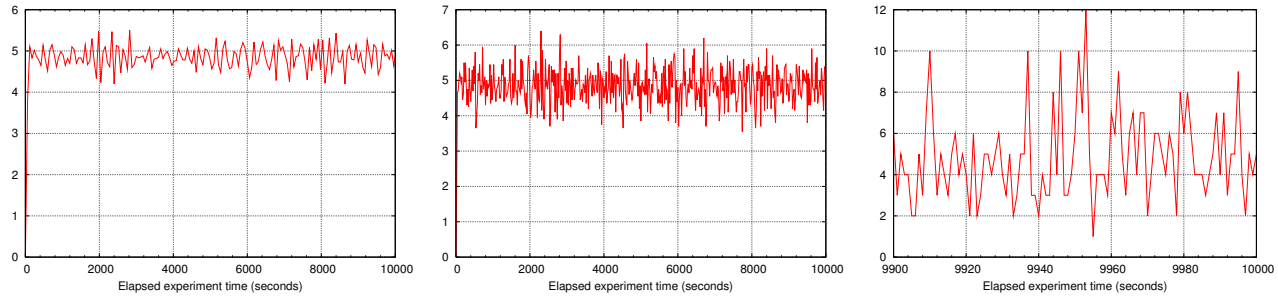


Figure 10.7: Customers arriving pr. 60 seconds (left), pr. 20 seconds(middle) and pr. 1 second in shorter interval (left)

By observing the CPU utilisation on the web node in Figure 10.8 it looks like the system is in steady state already after 550 seconds. This makes sense since all nodes are running full load by that time. See Section 8.6. Figure 10.8 also shows the utilisatin in a shorter interval, and it varies between 60% and 75% CPU utilisation.

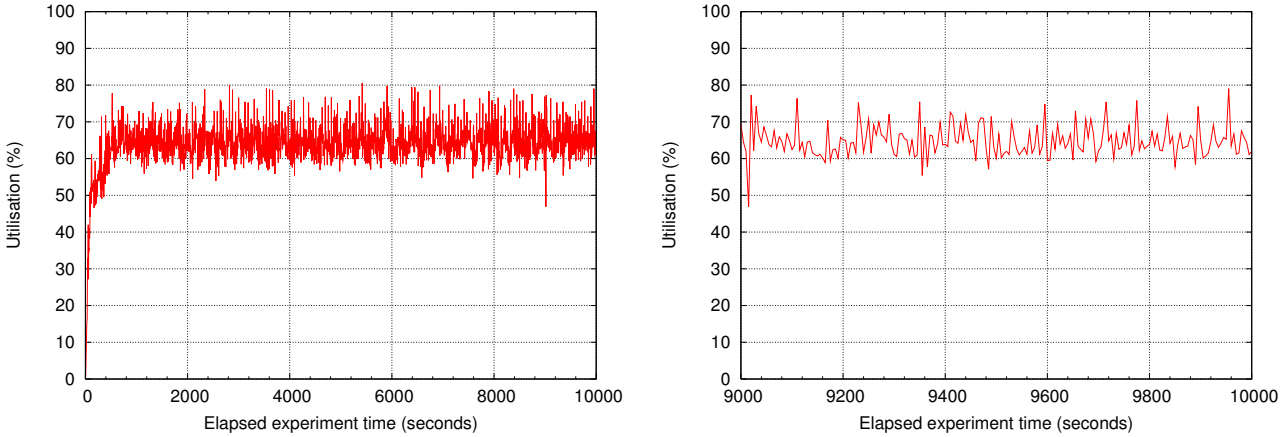


Figure 10.8: Web node CPU utilisation. Figure to the right shows the data in a shorter interval.

10.7 Steady state: long runs

A long run measurement was performed to check whether the system actually were operating in steady-state. The baseline system is measured with 1000 concurrent user sessions. The measurement started around 18:00, and was ran for 12 hours.

10.7.1 Web server node stats

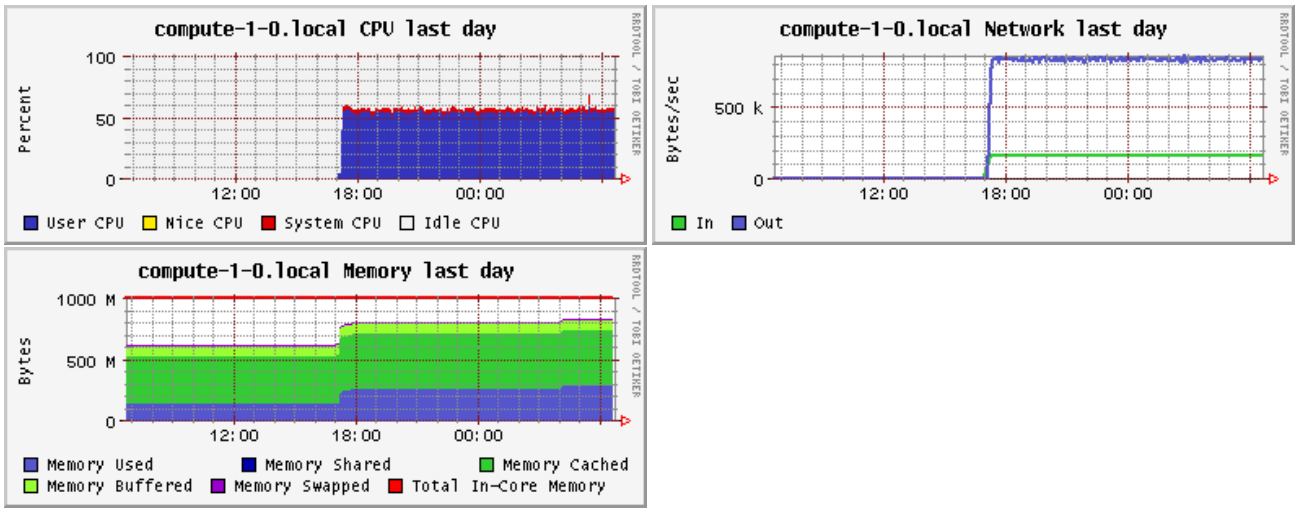


Figure 10.9: Web server stats CPU, Network, Memory

Both CPU and LAN usage is very constant. A small increase in memory usage can be seen at approximately 04:00. Since the available Java memory is predetermined as heap size, the memory increase might be due to operating system events.

10.7.2 Application server node stats

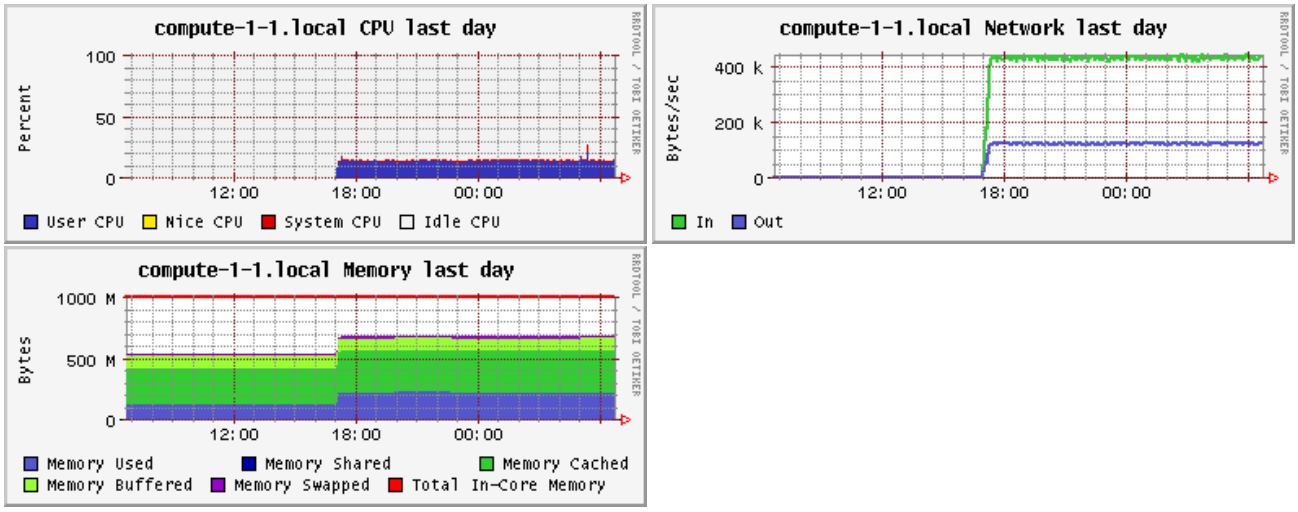


Figure 10.10: App server stats (a) CPU (b) Network (c) Memory

Utilisations are very constant for the application server as well.

10.7.3 Database server node stats

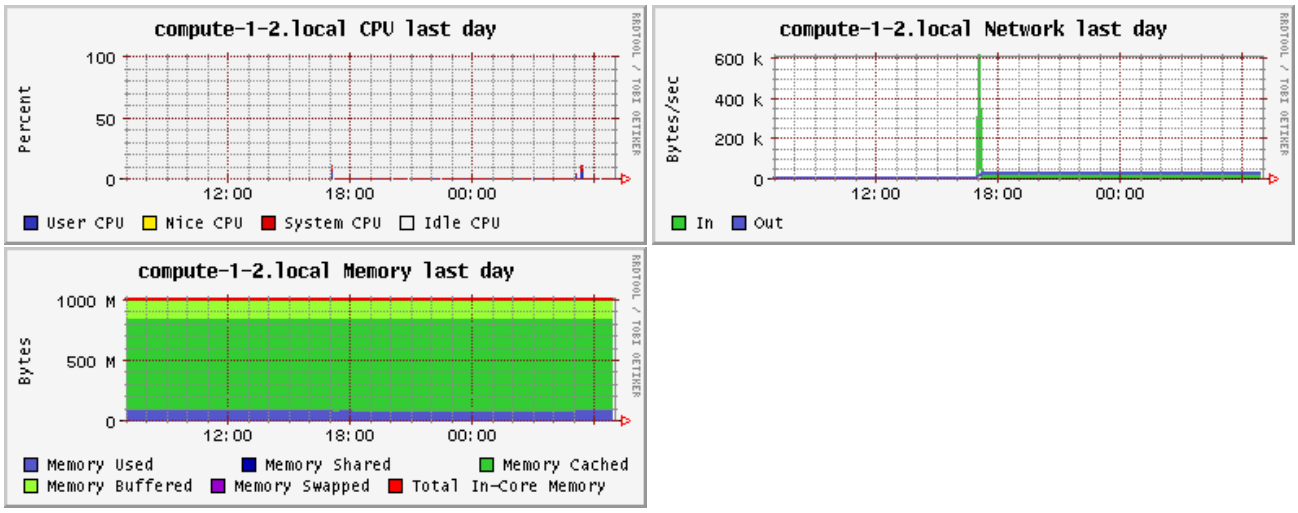


Figure 10.11: DB server stats (a) CPU (b) Network (c) Memory

We can hardly spot any CPU usage on the database CPU. Memory usage is very constant. The spike at 18:00 for network utilisation is due to database startup, when files are copied to the database server.

10.8 Steady state: garbage collector investigation

The garbage collector performance is calculated by analysing log files after measurements. See Section 2.7 for a description of how to produce the logs. Section 9.9 shows the results from the toolbox analysis of the log files.

We want to check if the garbage collector really is steady 550 seconds after experiment startup. Measurements are performed on upgrade nodes with a heap of 700MB. Figure 10.8 shows the garbage collection activity. It shows how much of the total CPU usage is due to garbage collection. The graph also shows the heap usage. The CPU usage is accumulated in 1 second intervals. This means that CPU usage from several minor garbage collections are accumulated into one sample. The heap is plotted each time a garbage collection occured. There were several minor collections each second.

From Figure 10.12 one can read that the heap usage stabilises after approximately 3000 seconds. Garbage collector CPU usage also remains stable after that.

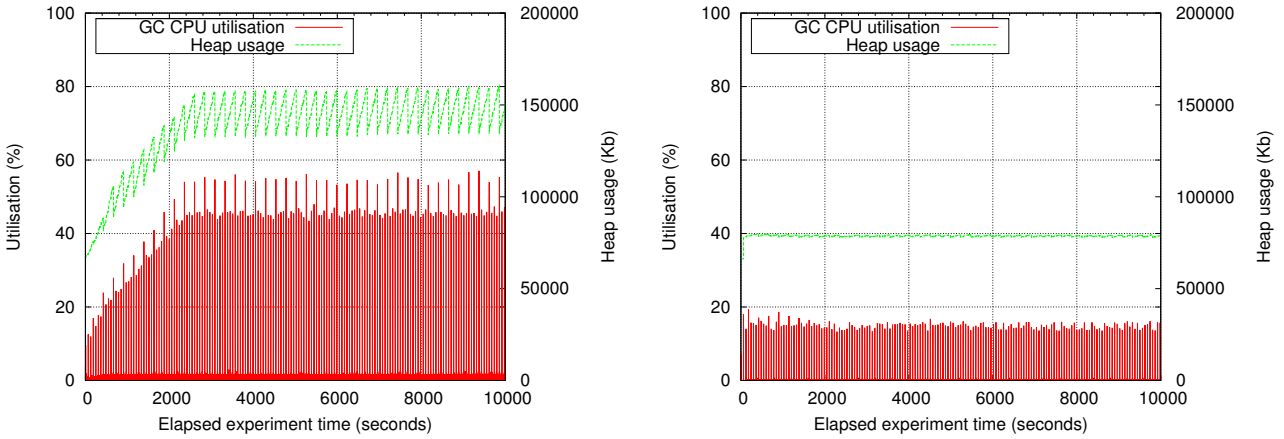


Figure 10.12: Garbage collector activity Web node (left), App node (right)

From Figure 10.12 it may look like a lot of the CPU time is used for garbage collection, but the fact is that 2.7 hours of measurement data is compressed into one graph.Figure 10.13 shows a more representative picture of garbage collection CPU usage. Full garbage collections are performed each 60 seconds, which takes approximately 100 milliseconds in the startup phase, and up to 600 milliseconds at time t=3000 seconds

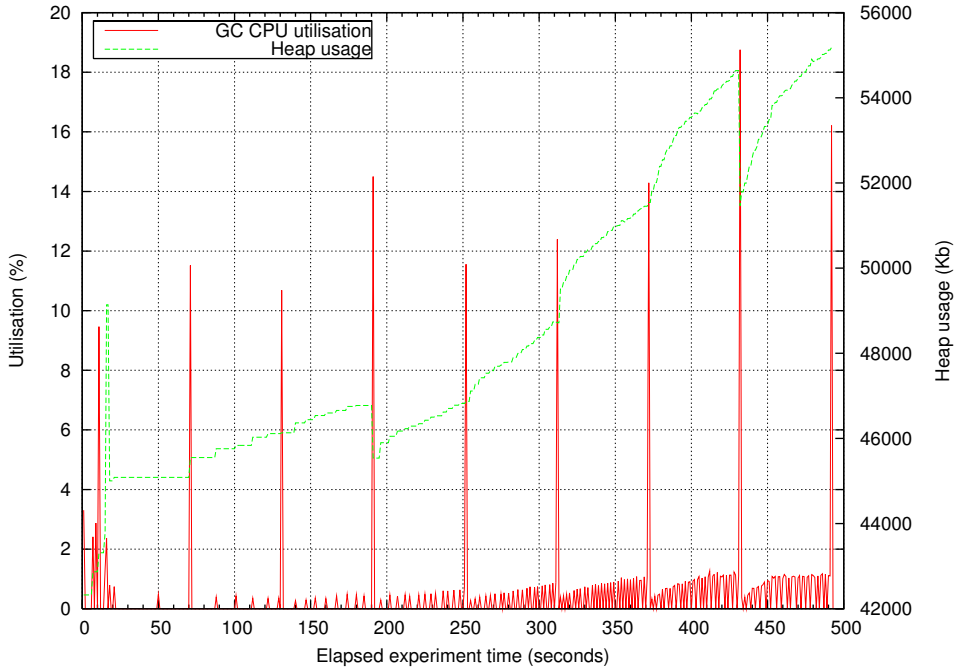


Figure 10.13: Closer look on garbage collector activity

10.9 Experiment problems

This section sums up some problems between the measurement results on baseline nodes versus the upgraded nodes. In the next chapter we will show that the effect of the problems is neglectable for our scope. But even though we consider the effects to be neglectable, we want to show how to deal with such problems.

As explained in Section 6.3, the baseline nodes were not available on Clustis2 throughout this project, so new and improved measurements could not be performed on the baseline nodes. This combined with the fact that some result files from baseline measurements on Clustis2 were lost, resulted in some complications:

- We discovered the fact that the system was not fully stable after 9 minutes, see later in this chapter: Section 10.8. Pilot measurements showed initially no noticable increase in the utilisation after 9 minutes. But improved measurement design and graph visualisation showed that the garbage collector was not stable until 3000 seconds out in the experiment. The reason why it was not detected is that the garbage collector CPU usage was never more than 2-3% of the total CPU usage under maximum load, and at that point the CPU was fully utilised.
- It was not possible to repeat the baseline measurements. But key measurement results are available, making it possible to validate the baseline model. By comparing the measurement designs with the available results, we try to interpolate the results. Here are the key measurement results that are available from baseline nodes on Clustis2:
  - Validation at operating point N=1000, heap=400MB and user base=50K, where data was analysed in the interval <550,1150> seconds in the experiment.
  - Parameter measurements: service demands for all requests in the workmix. The runlength of each measurement was 10K. See Table 13.1.
  - Parameter measurements: database service demands for increased user base load. See Section 13.4.

The consequence is that that all baseline measurement results are obtained in the interval <550,1150> seconds, while the upgrade measurement results are obtained in the interval <3000,5000>. In Section 10.10 these two designs are compared to eachother.

At the time when the baseline measurements were performed on Clustis2, the toolbox scripts that automates the task of creating graphs from measurement data were not created. This is why we did not discover that the garbage collector had not reached steady state after 550 seconds in the experiment, since the system looked very stable according to CPU utilisation and response time results. After visualising the garbage collector activity it became clear that the system was stable after 3000 seconds of elapsed experiment time. All measurements on upgrade nodes were from now on measured in the interval <3000,5000> seconds instead of <550,1150>.

Measurement scripts were improved also before measuring the upgrade nodes, and some of this info is not available from baseline measurements:

- More GC info: Number of GC collections, number of full GCs, avgerage and max heap size
- More Grinder info: Automatically calculated user session throughput, number of samples of each test, ratio between tests, automatically calculated workmix and thus automatically calculated session response times.
- Some of the GC log entries were not registered, so a few garbage collection entries were not recorded, resulting in a slightly lower GC service demand.
- Utilisation logs around midnight were not correctly analysed. The reason is that Sysstat only registers time and not the date in the logs. The new utilisation scripts were altered to be able to take into account when the time changed from 23:59 to 00:00.

10.10 Comparison of old and new interval design

We want to find the effect of increasing the old measurement interval from <550,1150> seconds, to the new measurements using the <3000,5000> seconds interval. We also want to compare the old

scripts and old kernel versions used for the baseline measurements with the new experiment setup.

For this task only upgraded Clustis2 nodes are compared to each other. We tried to find comparable results for a load around N=2500, and a heap of 400MB. The closest match were heap sizes of 200 and 600. Two old measurements were found suitable, heap=600 (N=2400), heap=200 (N=2520) and. Both results are obtained in the <550,1150> interval. The old results are then compared to new measurement results from the new design in the <3000,5000> interval.

Interval	Load	Heap	Web	App	GC_{web}	GC_{app}
<550,1150>	2400	600	142ms	40ms	2.6ms	0.67ms
<3000,5000>	2400	600	136ms	38ms	3.2ms	0.65ms
Ratio					1.2	
<550,1150>	2520	200	-	-	3.2ms	0.85ms
<3000,5000>	2525	200	-	-	4.1ms	0.77ms
Ratio					1.3	

Table 10.1: Comparing service demands for the two interval designs

Table 10.1 shows service demands per user session for the two intervals. The effect of the new interval is hardly noticable: the Web server garbage collector service demand has increased by 1 millisecond, while the service demands for Web and App server actually have decreased a bit for the new interval.

10.11 Choosing the operating point

With the results from the pilot measurements, we have enough information to choose the operating point.

As CPU utilisation and overall contention increases, quality of service is degraded. The operating point should reflect as much load on the system as possible, without too much contention and reduced response times. The task of formalising the finding of the operation point is considered further work.

To find the baseline operating point, various performance indices are checked. If the operating point is below some threshold, effects from operating system and hardware contention can be disregarded. The goal is to find the highest utilisation we can run the servers on, while the effects of these aspects do not exceed some threshold.

- **Congestion**: The operating point cannot be set too high, where congestion effects degrades the response time too much.
- **Reserve extra capacity for bursts**: The devices should be able to handle short bursts in the load.
- **Response time:** The response time should not be too high. Using the pilot measurements in Figure 10.4, we choose an operating point before the graph reaches its knee and then grows exponentially.
- **Paging**: If memory is never fully utilised, paging will not occur, and this aspect can also be disregarded from the analysis. In the Java domain, one can limit the maximum heap size available to the JVM, so that the main memory never gets full.
- **Garbage collection:** By configuring the system in such a way that there is always available memory, garbage collection effects can be kept to a minimum.

- **Context switching:** Keeping the load under some threshold, context switching can be disregarded

When deciding the operating point, we considered mainly reserving capacity for bursts and keeping the delta of the response time low. For a load of N=1000, the response times still increased slowly and there were extra capacity for bursts. The operating point was fixed to 65%.

11 Investigation of non-linearities

This goal of this chapter is to search for sources of non-linear effects. A few possible sources were identified by Fagerlie-Landmark, and the possible sources are investigated here.

We will describe the challenges of measuring non-linear effects in Section 11.1.

In this chapter we will mainly measure load-dependencies on the upgraded system. First we investigate the system as a whole with multiple user measurements. Then we investigate the software components separately with specialised measurements.

We will obtain some results for non-linear effects. We show that paging is non existent, and that paging is therefore no source. We will then show some results for garbage collection.

11.1 Load dependence vs non-linearity

To investigate non-linear effects we must consider at least two systems with different sizes. Measurements on those two systems are needed to quantify non-linear effects. When we are increasing the load without increasing the system size, we are only analysing load dependent effects, rather than non-linearities as we have defined it in Figure 3.1.

In a developer context the upgraded system may not be available, and the developer can then only investigate non-linear effects analytically. If the developer has a baseline system to measure on, his best option is to measure load dependent effects and try to deduce whether these are sources for non-linear effects.

We will to start by looking for load dependent effects. These are effects are dependent on run-time variables. We measure the effect on resource usage when we increase the load or dataload. This is performance evaluation and not scalability analysis according to our definition. We assume that there is some connection between load dependence and non-linearities.

11.2 Static model and load dependent variables

Before creating a model it is important to know something about the system. Experience, code analysis or measurements must be used to decide where to put the modelling effort. We perform measurements to search for non-linear effects. The findings of such an investigation should determine where to put the focus.

The SP model captures static properties of the system. But even though the model is viewed as static, some static properties may be dependent on run-time variables. Because of this, we have to know something about the system under live load in order to obtain parameters to the static model. Parameters dependent on run-time variables are measured in Chapter 13.

Another consequence of static properties being dependent on run-time variables is that the model focus should be concentrated to components with these dependencies. To reduce the amount of work obtaining parameters, all other components should be as simple as possible. Therefore we want to determine where there are non-linear effects (if any), before we create the SP model.

11.3 Groups of non-linear effects

There are three groups of non-linear effects:

Congestion effects Congestion effects influence all parts of the system. When multiple entities compete for shared resources (CPU, disk, network) there will always be contention to some degree. Even on the software level there is competition for resources. SAM tries to eliminate the effect of congestion in the analysis with the concept of operating point.

Software effects There are two types of software effects: Workload and dataload dependent effects.

Platform effects Subsystems are used to support processes at a particular level. To increase the platform resource, on can either replicate or upgrade the system. Replication of subsystems may introduce overhead as a consequence of data consistency maintenance. We do not analyse replication in this project, but we do analyse the upgrade of subsystems.

11.4 Possible non-linear sources

We have identified some sources for possible non-linear effects that we want to investigate:

- **Garbage collection:** The garbage collection is dependent on:
  - Load: The speed which new objects are created.
  - Amount of required heap: which depends on the number of concurrent users and their data-load.
  - Possible fragmentation of memory when heap is filling up.
- **Connection pooling:** The application server manages a pool of connections to the database server, so the EJBs can share the connections. A possible a non-linear effect might be introduced when the load N gets very high and the pool size is very large. Managing a large pool might lead to non-linearities.
- **Paging:** Paging may be a source, since disks are much slower than RAM. If paging occurs, the system throughput and response time might be severely reduced.

11.5 Service demands on upgraded nodes

Multiple user measurements were performed on upgraded nodes to obtain the service demands. The load is defined by the workmix script that implements ser sessions. Figure 11.1 shows the calculated service demand on each server per user session. The graph shows the results from ten measurements. We can read that the total service demand for a user session at a load of N=2500 is approximately 138 + 39 + 15 = 190 ms. Service demand does not increase noticeably as load increases, so there are no obvious non-linearities in the system. Special measurements for the application server is presented in a later section.

Service demands are obtained by gathering statistics in 2000 seconds in the interval <3000,5000>. The number “logout” operations reported by grinder is the number of completions C. The throughput X is then C/2000. The service demand on the web server equals the mean web CPU utilisation divided by the throughput. $S_{demand,web} = \frac{U_{web}}{X_{web}}$. The calculation is repeated for both App and DB server.

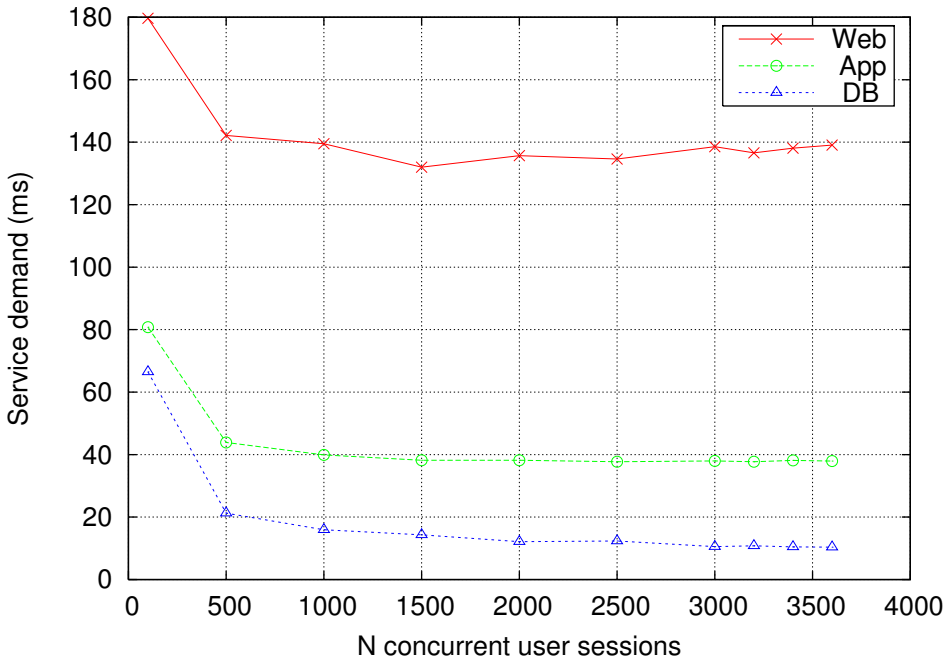


Figure 11.1: Upgraded system: Service demand per session. Heap=700MB

The system is also stress-tested with decreased heap size. Figure 11.2 shows the service demand results when decreasing the heap from 700MB to 175 MB. This will stress the garbage collector, resulting in higher activity due more frequent garbage collection. The heap memory gets full when the load is higher than N=3150. The JVM running Tomcat reports outOfMemory from that point.

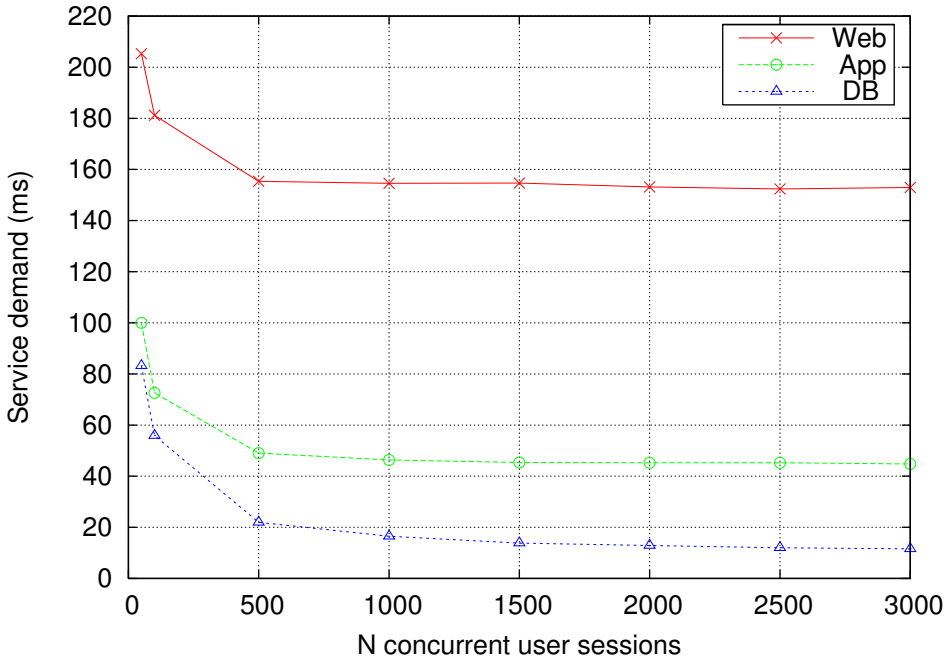


Figure 11.2: Upgraded system: Service demand per session. Heap=175MB

The measurement results from the heap=175MB series looks smoother than the results for heap=700MB.

This may be due to interfering processes. Some measurements for heap=700MB were repeated, and the graph became as smooth as for heap=175MB. The original graph is kept since it is sufficient.

11.6 Garbage collector service demands

We do not need to run separate measurements to obtain statistics about garbage collector activity. We use the data from the measurements performed in the previous section.

Figure 11.3 shows the garbage collector service demand for each user session on the web server. Three measurement series are plotted, for 175, 350 and 700MB heap sizes. The service demand per session is high until the load reaches N=1000. There is a small increase in the service demand when the load increases to max. Note that for heap=175MB the memory is fully utilised at N=3150, and we cannot push no more load on the system. Since the operating point on the upgraded system yields a load of N=2500, we see from the graph that there is only a tiny increase from the minimum service demand. Besides, the upgraded heap is set to 700MB, so we decide to ignore this tiny non-linear effect.

Remember that the garbage collection service demand a part of the total service demand that was measured in the previous section.

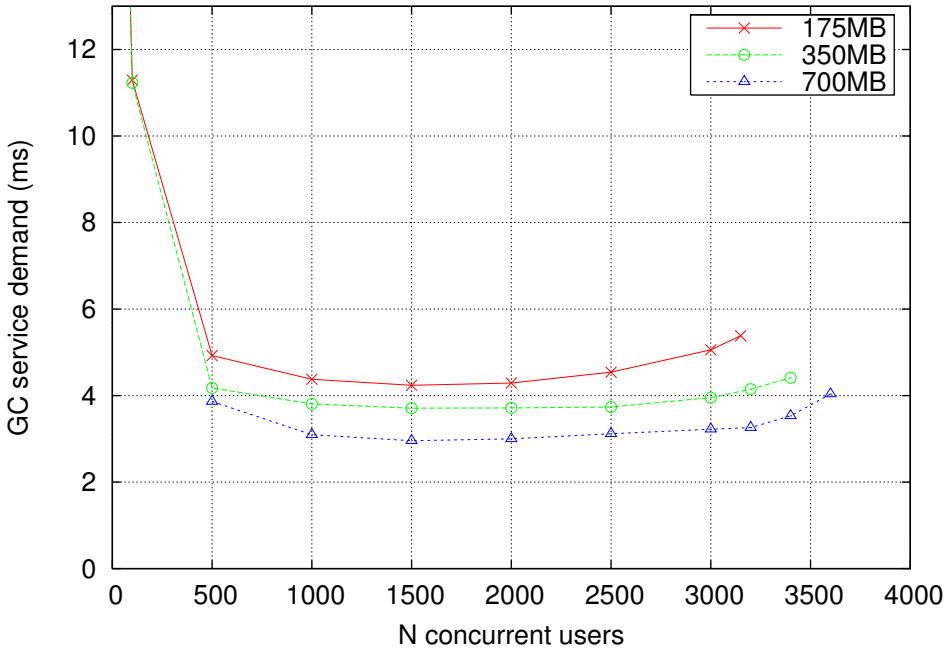


Figure 11.3: Upgraded web server: Garbage collector service demand

Figure 11.4 show garbage collector service demand on the application server. The service demands are decreasing all the way. For the measured range there there are no sub-linear effects on the application server. We are actually observing a super-linear effect. The downside is that the effect is only about milliseconds. The application server CPU utilisation is only around 25% for a load of N=3400. It looks like the graph is flattening for CPU utilisations larger than 25%, so the decreasing service demand for load less than N=1500 may be viewed as a system reaching steady state.

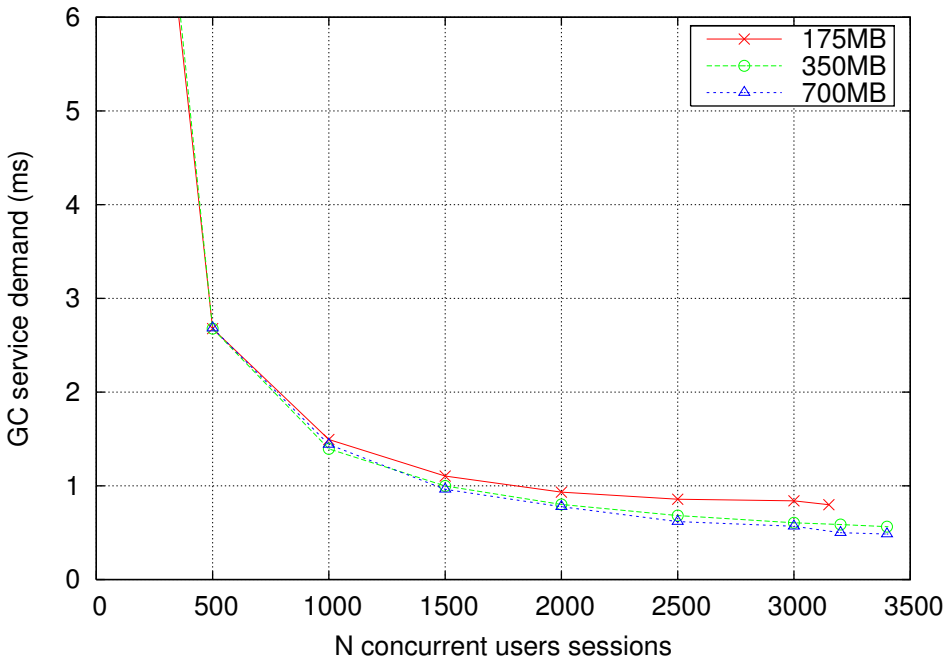


Figure 11.4: Upgraded app server: Garbage collector service demand

11.7 Garbage collector non-linear effects

In Section 13.3.2 we present the results of obtaining a function for the garbage collector resource demands. Table 11.1 is an excerpt from Table 13.2. It looks like the garbage collector shows a slight super-linear effect on the web server and the application server. The ratio is higher than the scale factor of 2.4 that we find in Table 15.5. The results must be taken with caution, since we only talk about milliseconds, and the total service demand for a user session is approximately 185 milliseconds on the upgraded system, and 430 on the baseline system.

				Service demands	
Node	Interval	Load	Heap	GC_{web}	GC_{app}
Baseline	<550,1150>	1000	400	8.5ms	3.7ms
Upgrade	<3000,5000>	2500	700	3.1ms	0.62ms
Ratio				2.7	6.0

Table 11.1: Garbage collector service demands for baseline and upgraded system

11.8 Application server focus

In the baseline configuration, the application server CPU utilisation never exceeds 25%. In this section we study what happens when the load is increased. To be able to increase the load on the application server, four web servers are connected to the application server. The modified baseline configuration is described in Chapter 6. The load is sent from the load generator nodes to the Apache load balancer node, which balances requests to the web nodes.

The measurements were performed on baseline nodes on Clustis2, and we only present the application server results. Figure 11.5 shows the utilisation as the load increases. We can see from the graph that we are able to utilise the server beyond the load of the baseline configuration.

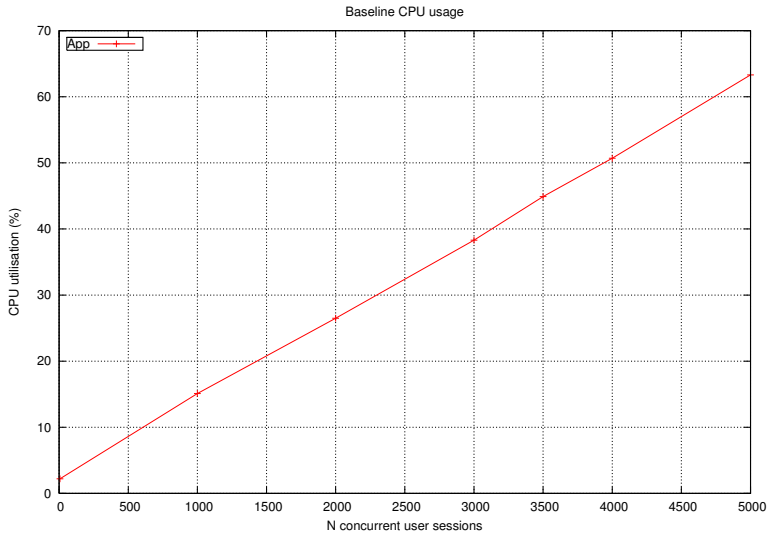


Figure 11.5: Modified baseline: Application server focus.

Figure 11.6 shows service demand per user session on the application server. The service demand does not increase even when the application server reaches 65% CPU utilisation. Note that these measurement results were obtained in the interval <550,1150> seconds, and not the new interval <3000,5000>. We believe that the results can be used as an indication that there are no non-linear effects for the system as a whole.

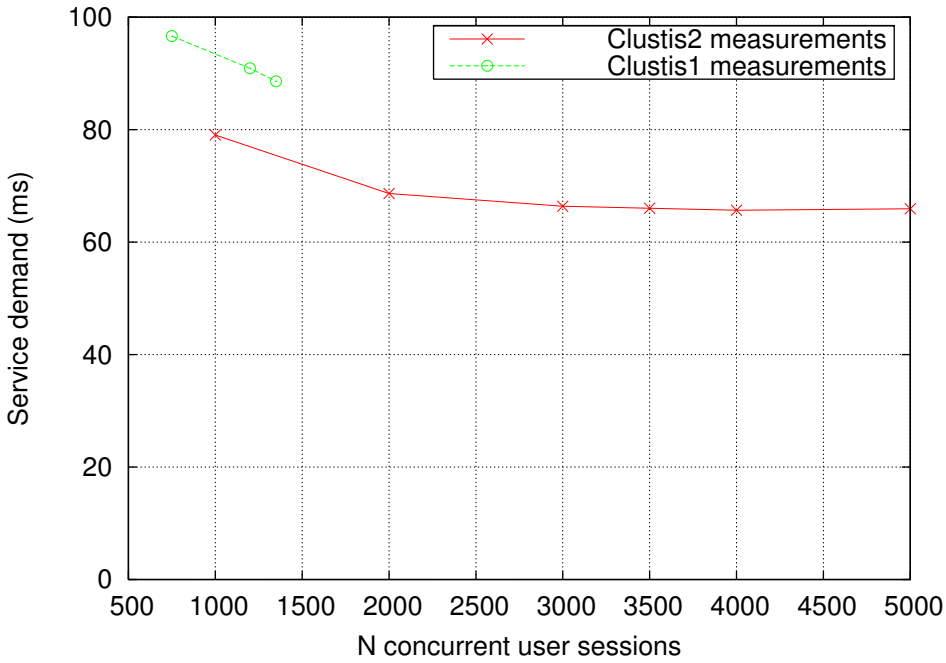


Figure 11.6: Baseline application server: Servide demand pr. session

Figure 11.7 shows the garbage collector service demands per user session. Both graphs also contain

results from baseline mesurements on Clustis1. The reason for that is to illustrate the effect of changing to an improved kernel and a newer operating system environment on Clustis2 compared to the older Clustis1.

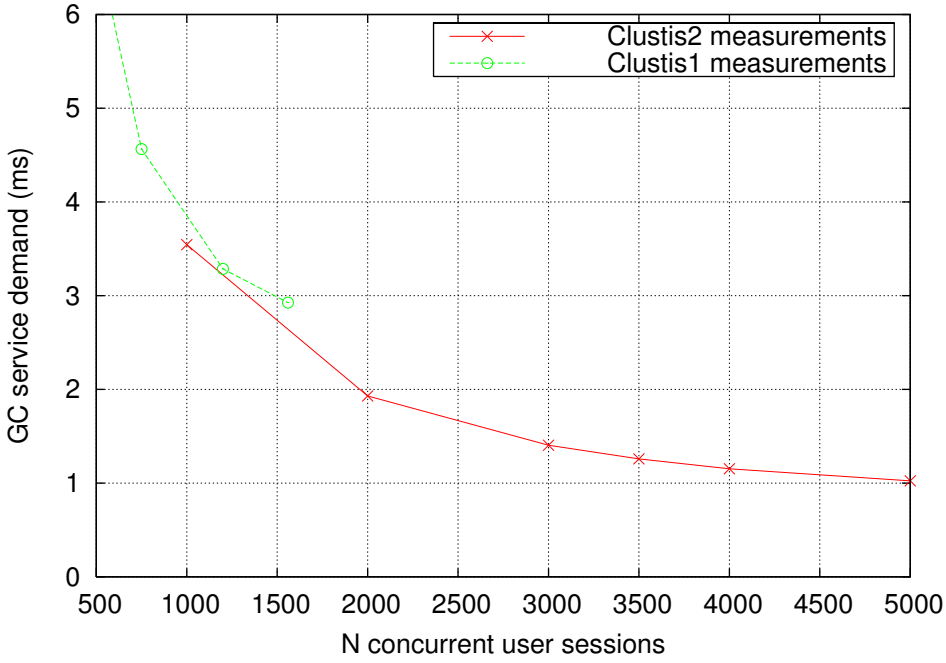


Figure 11.7: Baseline application server: Garbage collector service demands

11.9 Paging

Paging will not occur until the memory is almost full. To simplify the measurement analysis paging were avoided by not utilising all memory. Since all Clustis2 nodes has 1GB of ram, heap size was chosen to be 350MB for baseline and 700MB for upgrade, leaving some space for the operating system. The Java servers cannot use more memory than is assigned by the maximum heap size, so therefore paging will not be a factor to consider. Even when stress-testing the system with high loads that saturated the CPU on the upgraded system, heap memory usage was never higher than 200-300MB.

Some pilot measurements were done to check that these assumptions were correct. There were minimal paging activity in the stress-test case, so parameterising measurements for paging were cancelled. The paging activity was relatively constant, indicating operating background processeses were responsible.

Paging is measured while running multiple-user measurements, and activity info is extracted from the Sysstat logs. Figure 11.8 shows the paging activity for a load of N=3400 and a heap of 900MB. The web server is represented in the left figure, and the application server in the right figure. The paging info is shown as the mean in 180 second intervals.

The graphs show that no pages were read from disk. A stream of approximately 10 KB/s were written to disk or to operating system buffers. We conclude that for this project, paging is not a measurable non-linear effect.

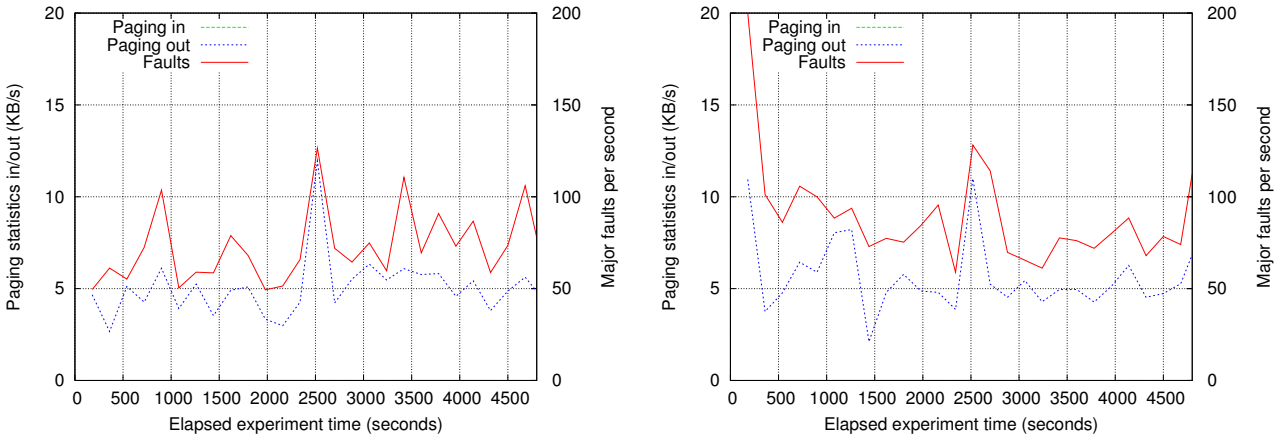


Figure 11.8: Upgraded web server(left) and app server(right): Paging statistics for heap=900MB

11.10 Connection pooling

Connection pooling is another possible source for non-linear effects. Connection pooling means that a pool of open database connections are created on the application server. The reason to pool the connections to the database server is that creating connections on the database server is expensive. Each connection consume memory and other resources. See [14] page 82 for a simple experiment that visualises the benefits of resource pooling.

Ideal case

If a system is upgraded by strict and uniform scaling, ideally the same amout of connections is needed on both the baseline and the upgrade: Consider a upgraded system S_1 that has twice the capacity of the baseline system S_0. The system consist of both web-, application- and database server. Since S_1 has twice the capacity of S_0, it follows that it has half the response time of S_0. The load on the database server is doubled from the baseline case, but the response time is half of the baseline. The result would be the same number of connections. This analysis is only valid in the strict case. If the service demand pr. transaction increases on the database server as a result of increased load, each transaction takes more time and will therefore hold the connection longer. More connections are needed, and one may notice non-linear effects.

Analysing the logs

We want to obtain the mean number of active connections for a given load. We have two possibilities for investigating connection pooling, by analysing either JBoss or the MysSQL logs. The JBoss log only notifies when connections are created, but it does not notify when connections are removed from the pool, and it only notifies that some cleanup was performed. So JBoss does not give us enough information.

The MySQL log file on the other hand, relates connection ID to the SQL query. Connection number 5 and 6 are active in this example.

Listing 11.1: SQL log

```
060629  6:52:37     5 Query   SELECT ID,FIRST_NAME,LAST_NAME,STREET,ZIP, CITY, PASSWORD FROM CUSTOMER
                                WHERE ID=18862548989
                    5 Query   SELECT * FROM ACCOUNT WHERE OWNER=18862548989
060629  6:52:52     6 Query   SELECT * FROM PAYMENT WHERE OWNER=19256091773
```

A simple approach is to divide the queries in partitions of 4000. We then count the unique connection IDs for each partition. That will give us the number of active connections for each partition. We calculate the mean of all partitions and obtain a mean for the overall connection pooling.

We performed multiple-user measurement with a load of N=3600 user sessions. The JBoss node shows 26% CPU utilisation, while the Tomcat node is almost fully utilised at 96%. The database server shows a utilisation of approximately 5%.

A script was created to calculate the mean number of active connections. The result is shown in Appendix, Section B.1. The script calculated that only 3 connections were active for each partition of 4000 SQL queries.

Measuring 100 idle connections

A multiple-user test was performed on the upgraded nodes, with standard work-mix and a load of N=2400. JBoss application server was forced to open 100 connections in this experiment. As expected, there was no increase in the application server utilisation when having 100 almost idle connections in the pool. From the MySQL log results, we know that there are only 2-3 active connections for such a load.

Number of connections in pool	Default(5-15)	100
Application server CPU utilisation	19.5%	19.3%

Table 11.2: Measuring 100 idle connections

Conclusion

Assuming that the 3 active connections were fully utilised, a load of N=10800 (3 * 3600) would require 9 connections. It is hard to imagine any non-linear effects introduced at such a small pool size. A rule-of-thumb indicates that 20-60 connections in the pool is common for most applications. In large enterprise domains however, these numbers may increase a bit.

We conclude that we cannot stress the system enough to identify any connection pooling non-linear effects.

12 SP model construction

This chapter describes how to create an SP model in a Java environment. We present some rules of thumb from the work of making an SP model of BankApp. We touch both the process of aggregating classes into components, and also the process of how to identify the component operations.

12.1 Rationale for simplicity

The design goal of a model is to have as few components as possible, while keeping granularity fine enough to be able to express non-linear effects between components. Since every component needs to be connected to at least a processing device, the number of edges grow fast when adding components, thus one want to achieve as few components as possible. Each edge is represented by a matrix, and all matrices must be parameterised. Parameterisation can be a very time-consuming task, so one want to keep the number of matrices as low as possible.

The other goal is to achieve high cohesion and low coupling by grouping objects that interact a lot with each other, into one component.

12.2 The process

We start with one software component per node initially. This components consist of all software on the that node, including operating system, server software and application software. We can then decompose the model by considering these rules of thumb:

- Focus on important components that are expected to have scalability issues.
- Unknown, difficult or uninteresting parts of the system are lumped together in components. This effectively allows us to view these system parts as black boxes.
- Analyse source code and identify groups of classes based on design paradigms.
- Drill-down: Decompose a component when assuming that there are non-linear effects in a class or a group of classes. Remember to keep high cohesion and low coupling. Components that interact a lot should be in one component.

12.3 SP model construction

A lot of aspects must be considered when modelling a system in SP. This is a non-exhaustive list of things to consider when construction an SP model

Checklist for SP-diagram:

- All components must follow the general SP rules described in [5]. Denote all links correctly as processing, storage or communication.

- Top-down-view: The SP model must be a directed, non-circular graph, resulting in hierarchies of modules. Check that components use the other components in a top-down-fashion.

- One single component at the top-level. The component offer services to a user or maybe another system.

- Hardware devices are at the lowest level of the hierarchy, and must be placed in the bottom of the diagram. For example CPU, disk and LAN.

- Simplicity: Use as few components as possible.

- If component A uses another component B in a one-to-one fashion, aggregate the lower level component B into the component A.

- Hide components outside the analysis scope, or components that are beyond our knowledge to decompose and express them as a single component. For example: database servers are complex creations, so unless one wants to analysis database servers, it may be a good choice to represent it as a black box by using one component.

12.4 The BankApp SP model

Figure 12.1 shows the SP model used in this project. Since the software components are not decomposed, this is a general SP model that can be applied to most configurations that uses separate nodes for application and database tiers.

Since there is only one software component per node, the components represents all software on the node. These components can be treated as black boxes, and measuring their effects on the hardware is straight-forward, using system-level instrumentation to obtain the total resource usage.

The disk components on web and application server are omitted. Once the server software is running and the measurements are started, the software only uses memory and LAN. Paging is virtually non-existent on these nodes. The only source of disk activity required by the servers would be logging, but this is turned off. Logging info would normally be saved on a file server and not saved on a local low-performance disk.

Also note the hierarchical nature of SP: In a developer view, the client browser never directly uses the application server. The web server acts as a middle-man, sending requests to the application server when needed. SP allows the modeller to represent a more distributed scheme. If the client indirectly uses the application server, there is nothing with modelling the client as using the application server directly. All that matters is that work is mapped from higher levels to lower levels of software, down to the hardware devices.

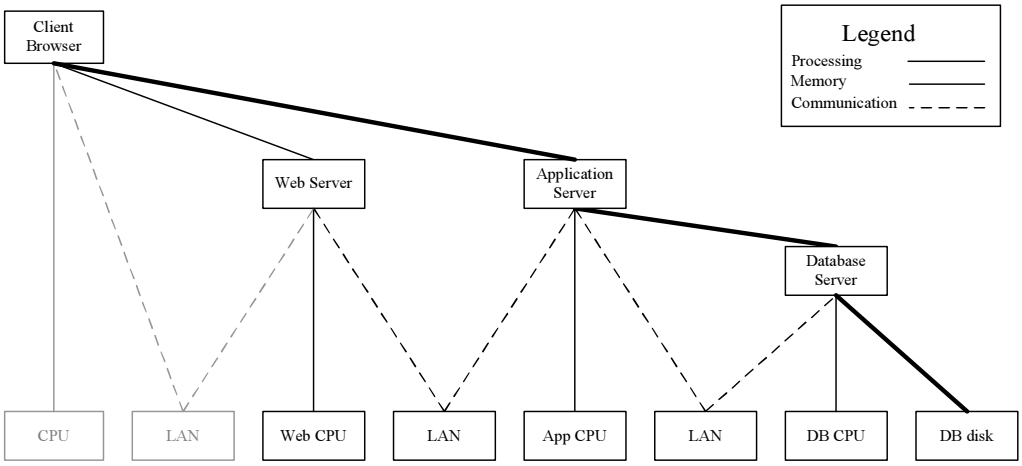


Figure 12.1: SP model

12.5 Operation types

Operations for each component in the SP model must be identified. Three types of operations are identified: Top level, software and hardware operations.

Top-level operations Operations accessible by the client, such as web pages or web services.

Software operations In a J2EE environment, components often contain Java classes. The methods for the entry point class of a component is used as operations.

Hardware operations Hardware device operations can be such as CPU instructions, memory words (read or written), disk blocks or bytes (r/w), or network messages (r/w)

Figure 12.2 shows how java methods are mapped to SP components. The figure shows that several classes can provide the component with entry-point methods. The entry points are viewed as SP operations. The figure also show that the component is composed of several classes, and that some of the Java classes uses operations in other components.

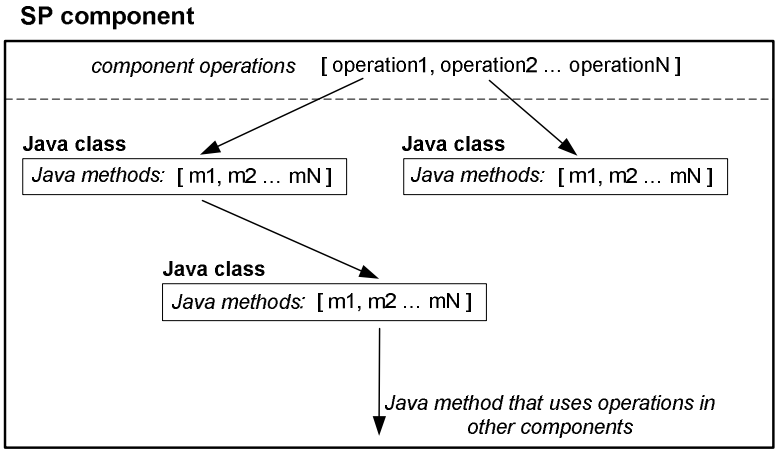


Figure 12.2: Mapping of Java methods to SP components

12.6 Complexity function investigation

Several approaches can be used to find the complexity functions. What to use depends on what is available of software and hardware, what can be measured and the experience of the modeller.

- **By source code:** The simplest functions are found by following the method calls in the source code. Each component operation are bound to a Java method. All operations can be bound to the same Java class, or different classes. Follow the method calls until a call is found to another software component. Such cases are parameterised as a constant, unless the code has non-linear issues.
- **Algorithmic analysis:** There may be more complex relationships involving algorithmic aspects. The source code and design specification must be investigated to estimate the effect of the algorithm in relation to run time variables.
- **Measurement:** Measurements produce more accurate parameters than estimation, at the cost of time needed to configure experiments. Load-dependent properties are hard to quantify without measurements, so this is usually the sound approach.
- **Profiling:** Java profilers can be used to capture component level resource usage. The static relationship between the classes can be obtained by using profilers.

13 SP model parameters

The preceding chapter introduced the construction of an SP model. Software was organised in components, and operations were identified for each component. The next step is to parameterise the model by finding the complexity functions. Complexity functions are elements in the complexity matrices, and the task is to discover the form of the functions, and quantify the function values.

Two aspects of the system needs parameters: Static resource usage measured with single user measurements, and dynamic resource usage measured with multiple user measurements.

We are going to establish a dataload(heap) dependent function for garbage collector resource usage. This is not an SP complexity function since the unit is service demand. A complexity function denotes devolved work, which is unit-less. The reason is that we are using the function directly in the system level model (dynamic), and not in a component based model (static).

13.1 Calibration vs. parameter measurements

When the model is fully parameterised, it is time to calibrate the model. The output of the model is calibrated with measurements. The total resource usage (service demand) output from the model must be compared to actual total resource usages found by measurement. With single user measurements the CPU utilisations are found for each request with system-level instrumentation.

Ideally, we would create an SP model, parameterise with data from code inspection, rule-of-thumb performance data and specialised measurements. Then parameter measurements would be performed to decide the resource usage for each operation in the workmix. The parameter measurements would be used to calibrate the SP model.

Instead we use the parameter measurements to parameterise the model directly. The reason is that this project revealed no significant non-linear effects in the range that could be measured, so a simple SP model was used. Since the system is ready for measurements, this procedure is cheaper than analysing code in terms of amount of work.

As mentioned before, the web, application and database tier will remain separate entities. The gain is that each server component can be treated as a black box. This means that the effects from different software components do not need to be separated. The resource usage for each operation is the sum of resource usage by all software components in the system, including all Java method calls, operating system effects and Java Virtual Machine.

For example: If the web tier consisted of two or more components, we would have to separate the CPU contribution of each component, then estimate the complexity functions, and finally use the parameter measurements to calibrate (adjust) the complexity functions.

Since one component contains all software on a node, the node *is* the component, and parameters can be measured using system-level tools. When decomposing this component into two or more, profiling may be needed to separate the effect from the different software components.

13.2 Top-level operation parameters

The resource usage for top-level operations is captured by single user stepwise measurements. Only one load generator node is used to generate user requests. Think times are removed, so the server receives requests in a sequential and non-stop fashion. When the client receives the response after a request is processed, a new request is sent immediately.

13.2.1 Tests

Ideally we would want to measure each request separately. But some requests depend on others, a user must log in before he can access his accounts. The solution is to split a transaction and measure it each part separately. The service demands for a particular request can be found by subtracting the service demand for the preceding requests.

Consider determining the service demand for the Login request. First we find the service demand by measuring Init only (Test 1). Then we measure both Init and Login (Test 2). The service demand of Login is then the service demand of Test 2 minus the service demand from Test 1.

These are the tests we perform to determine the service demand for all eight operations:

Test 1: Init

Test 2: Init + Login

Test 3: Init + Login + Payment

Test 4: Init + Login + Payment + PaymentDetails

Test 5: Init + Login + NewPayment

Test 6: Init + Login + NewPayment + ConfirmPayment

Test 7: Init + Login + NewPayment + ConfirmPayment + PaymentReceipt

Test 8: Init + Login + Logout

Each of the tests are run 10 000 times in loop. For each repetition, a new user will be logged in. The load is represented by one Grinder thread:

1 Grinder thread = [$\text{testY}_{\text{user1}}$, $\text{testY}_{\text{user2}}$ ... $\text{test}_{\text{userM}}$]

13.2.2 Example of calculation

The service demand is calculated using the service demand law, see Section 3.16.

$Service\,demand\,D_i = \frac{U_i}{X_0}$ with respect to device i, and 0 denotes the entire system

The throughput for each test is found by calculating the the time it took the load generator to complete the test script. Dividing elapsed time by the total number of tests performed gives the throughput. Dividing the mean CPU utilisation gives the service demand. When measuring, we use the script *view-parameterising-summary.sh*, see Section 9.9 for an example of the script output. This is a modified version of the regular view-summary.sh-script. The script automatically calculates the time it took for the load generator to complete the test, and outputs the combined service demand for a test.

To find the service demand for Login on the web server, we simply subtract the service demand for Init from the measurement of the combined Init+Login.

Test 1: [Init]

Elapsed time = 107 seconds

Throughput = 10000 requests / 107 s = 93.4 transactions per second

Utilisation= 77.9 %

Service demand = 0,779 / 93.4 tps * 1000ms =8.3 ms

Test 2: [Init + Login]

Elapsed time = 698 s

Throughput = 10000 requests / 698s = 14.3 tps

Utilisation = 71.0 %

Service demand = 0,710 / 14.3 tps * 1000ms = 49.5 ms

Obtain service demands for each requests:

Init service demand = [Init] = 8.3 ms

Login service demand = [Init+Login] - [Init] = 49.5 ms - 8.3 ms = 41.2 ms

These calculations are also implemented in a spreadsheet, see Appendix A.

13.2.3 Results

Results from parameter measurements are shown in table Table 13.1. All figures are in milliseconds. The service demand for each node is shown (web, app, db), along with the total service demand on the system as a whole (system service demand). The table also shows which requests that need service from the application server. In BankApp, every EJB invokes the database server.

Multiplying the system service demand per request with the workmix gives the session service demand. The total service demand for all requests in the workmix is 427 milliseconds. This is called the user session service demand. 320ms is spent on the web server, 93 ms on the application server, and 13 ms on the database server.

Request	Web	App	DB	W+A+Db SD	Workmix	Session SD	Invokes EJB
Init	8,3	0,1	0,0	8,4	1	8,4	
Login	41,2	17,6	0,6	59,4	1	59,4	*
Payment	26,1	8,5	0,5	35,1	3	105,3	*
PaymentDetails	25,9	8,1	0,5	34,5	3	103,5	*
NewPayment	8,7	0,3	0,1	9,1	3	27,3	
ConfirmPayment	6,6	0,2	0,1	6,9	3	20,7	
PaymentReceipt	20,5	7,7	3,1	31,3	3	93,9	*
Logout	7,0	0,3	0,1	7,4	1	7,4	
Total						426	

Table 13.1: Parameter measurement results

For all requests that do not need EJB, we note some system or server background activity. Logout does not invoke the database(DB) server at all.

We initially calculated a service demand of 1,2 ms for the Logout request on the database server. This must be due to interfering tasks such as operating system background processes. We performed similar parameter measurements on Clustis2 which showed that there was no activity on the database server, so the value was set to 0,1ms.

13.2.4 Converting to instruction count

To parameterise the static model, a time-independent metric is needed. A naive approach is chosen, where one instruction equals one CPU cycle. The baseline CPU is 1400MHz,

Examples of instruction conversion for web server CPU:

Init: 0,00860s * 1400MHz = 12040 instructions

Login: 0,0411s * 1400MHz = 57540 instructions

13.3 Garbage collection and heap size

Since the baseline nodes are not available for measurements, results from upgrade nodes must be used to project the parameters to the baseline nodes. First we establish a function for garbage collector service demand, and then project it to the baseline, using available measurements from baseline nodes.

13.3.1 Garbage collection resource usage on upgrade nodes

Garbage collection must be taken into account when modelling a system. The garbage collection algorithms are quite complex, and it is not a trivial task to fully understand the inner workings of the garbage collector. Each JVM vendor is responsible for implementing the garbage collector as they see fit, making any generalisation hard.

Heap size has an effect on the garbage collector. A larger heap size means that objects can be collected less often. But at the same time there is a cost of maintaining a large heap. The key design consideration for GC-developers is throughput vs. response time. Several experiments must be performed to check the relationship between heap size and garbage collector in question.

Test description: For a given heap size, perform multiple-user measurements on baseline with load 1 to N, in steps of T. Extract garbage collector information and plot a graph. In such a graph we see how the garbage collector is affected by adjusting the heap size.

SUN JVM version 1.4.2 was chosen as the virtual machine to run the servers on, using the standard garbage collector.[1]

- Garbage collector is a property of the JVM. J2EE has no control over this. Some properties of the GC can be controlled at JVM startup time
- Different algorithms/implementations for different JVMs, hard to generalise

The garbage collector usage on the web server was investigated for a fixed load N=2400. Heap size was increased from 109MB to 700MB Figure 13.1. A trend line is also plotted: This shows how the service demand increases as the heap size gets smaller. The heap is fully utilised at 109MB.

The function behind the trend line is used as basis for the complexity function for garbage collection. The service demand on the web server is a function of the heap size.

$$f(h) = -0.003h + 4.9$$

[1] Since the servers are run on single CPU nodes, there should be no benefits from using the parallel garbage collectors. On multiple CPU nodes, the parallel collector would have shortened the "stop-the-world" pauses.

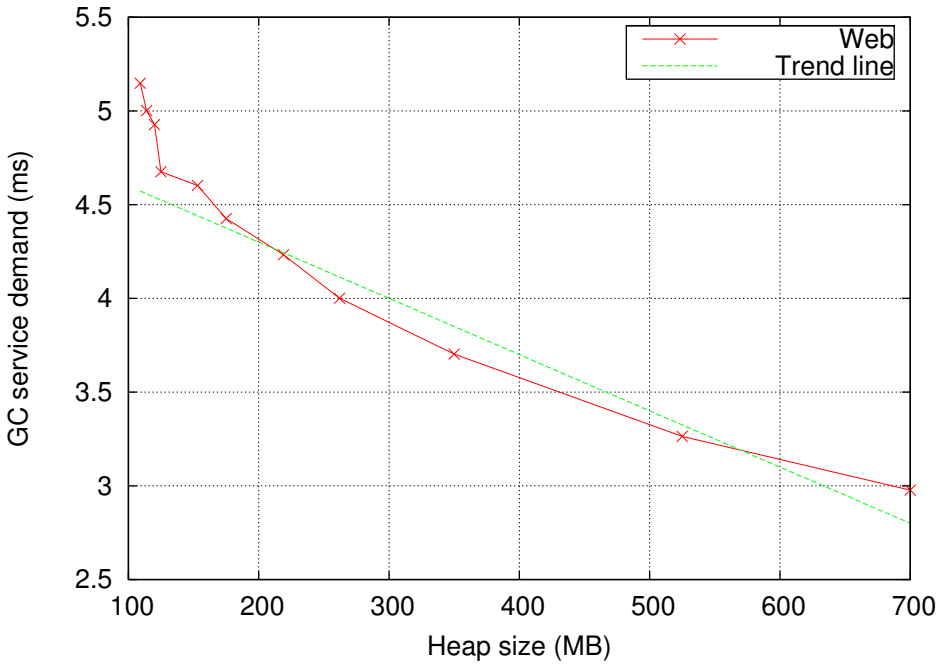


Figure 13.1: Upgraded web server: Garbage collector service demands when heap size increases

13.3.2 Projecting garbage collector parameters to baseline

In order to project the upgrade GC parameters to the baseline, a relationshionship between the two node types must be established.

Section 13.3.2 presents the results when comparing the baseline and upgraded systems. Using the baseline validation measurement at a load of N=1000 and heap=400MB, the GC service demand was 7.1ms. The upgraded web node yielded 3.1ms at a load of N=2500. Since the measurements were performed with different intervals, we use the factor found in Section 10.10 to project a new GC service demand for the web server. The adjusted service demand for the baseline web server is $7.1 \cdot 1.2 = 8.5ms$. The ratio is $8.5ms/3.1ms = 2.7$.

				Service demands			
Node	Interval	Load	Heap	Session_{web}	Session_{app}	GC_{web}	GC_{app}
Baseline	<550,1150>	1000	400	335ms	99ms	8.5ms	3.7ms
Upgrade	<3000,5000>	2500	700	139ms	39ms	3.1ms	0.62ms
Ratio				2.4	2.5	2.7	6.0

Table 13.2: Comparing garbage collector on baseline and upgraded system

The function is adjusted with the scale factor by multiplying 2.4 with the function. Note that we have already found garbage collection not to be load-dependent. The scale factor was found in Section 15.5.

The new function is then: $f(h) = -0.007h + 11.7$

baseline: $\frac{f(350)=9.24}{1} = 9.2$ The measured service demand was 8.5ms

upgrade: $\frac{f(700)=6.7}{2.4} = 2.8$ The measured service demand was 3.1ms

13.4 Increased database

While building a model of the database is out of scope of this project, end-effects must be accounted for. One interesting aspect to investigate is how the database behaves under changed dataload. Increased dataload means that the total number of users in the database is increased. The database files gets bigger, and thus the server must handle larger files.

Test description: Perform single user stepwise measurements on the baseline. Calculate service demands for all operations on the database server. Plot a graph of service demand over concurrent number of users. Repeat for increased sizes of the user base

Figure 13.2 shows the results of this test. Data-load is varied in the range of 50 000 to 600 000 users (50K, 200K, 300K, 400K, 500K, 600K). The task of generating a larger user-base is very time consuming, and for creating a larger user base. Modifications to the scripts must also be done to be able to handle larger databases. [2]

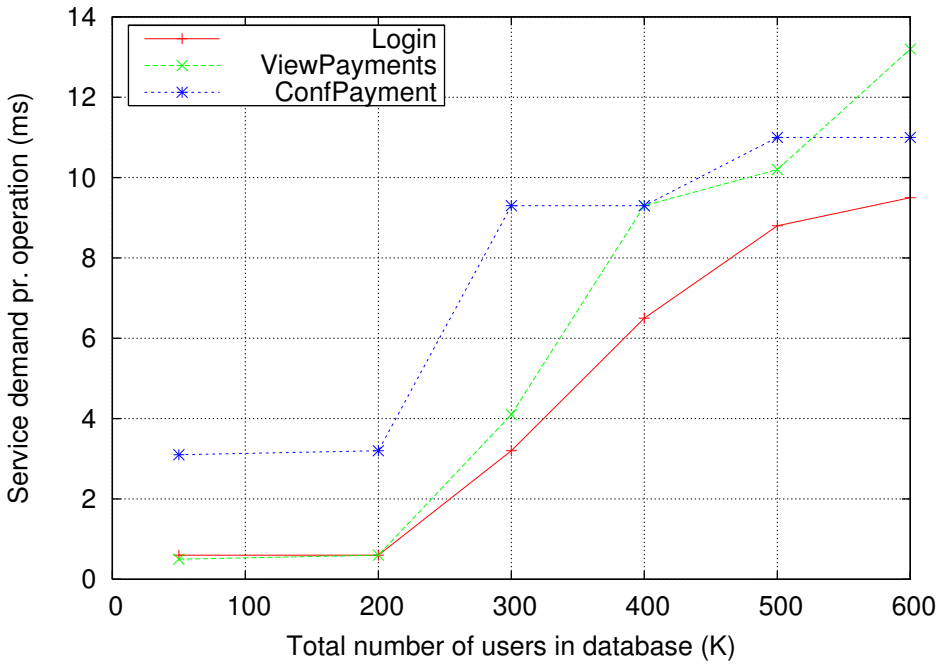


Figure 13.2: Database response time for increased data-load

These measurements show that the choice of a baseline database of 50K users do not result in increased service time when scaling up the system by a factor of 2-3. As long as we use a database with less than 200K users, no increase in service demand will occur. Since the baseline nodes were taken offline permanently a while after these measurements, no measurements could be performed to investigate this further.

Possible web and application server side effects As the user base is increased, we do not expect there to be any increased CPU usage on the web and application server, only on the database server.

[2]The reason is that the after the database is created, the data is read back from the database with a select statement. All data is read at once, so memory quickly gets filled up. For a heap of 800MB, we could transfer the data for 600K of customers. For bigger customer bases, scripts must be modified so that the select statement do not return all data in one transfer. This was not considered important to this project.

But to be sure two measurements are compared. Baseline measurements are performed with 1000 concurrent user sessions, and with a user base of respectively 60K and 300K users.

	User base	
	60 000	300 000
Web server	63,7%	63,2%
Application server	19,0%	19,0%

Figure 13.3: Comparing server CPU utilisations with increased user base

The measurements show that there were no increase in CPU usage as a result of increased user base. The results are as expected, since only the database should be affected of this.

13.5 Implementing the static model

Since we only have one run-time variable load dependent function, there is no reason to implement the SP model. [3]

We put the measured service demands directly into the dynamic model in the next chapter. We also add the increased garbage collector CPU usage is to the service demands.

[3] If we were to implement the SP model, we would either use a spreadsheet with matrices, or using the new SPlight tool mentioned in Section 1.6.

14 Dynamic model

The dynamic aspects of a system is how it behaves under load. Contention effects are simulated using a conventional queuing network. This chapter shows how to construct a dynamic model that represents the physical hardware devices. After the model is analysed, we compute the results, and it is shown how to combine the static and dynamic model to predict utilisation for the hardware devices.

14.1 Analytic approach

To solve the dynamic model, an analytic approach is chosen because of the simplicity of using an already existing spreadsheet implementation. The spreadsheet is a multi-class network solver implemented in an Excel spreadsheet by the authors of the book [15]. The spreadsheet can be downloaded from the authors. See the link [23]

Minor simplifying revisions are made to the spreadsheet so that the model can be parameterised with number of concurrent user sessions (N) instead of calculating request arrival rates. N is used to derive the arrival rate of each request according to the average user session duration of 515s. See Figure 14.4.

14.2 Open vs. closed queuing networks

Queuing networks can be modelled as open or closed. In an open queuing network, customers arrive at arbitrary moments, gets serviced and then leave the network. A closed queuing network on the other hand has a fixed number of users(or batch jobs) that never leave the network. The user gets serviced and waits a moment (think time), and then requests service again.

This chapter will show that both an open or closed model can be used to accurately predict CPU utilisations, but the open model will produce a slight lower total response time than the closed model.

The system boundary decides if the queuing network is closed or open. In an open system users arrive at the system and then leaves. In a closed system on the other hand, a user may get serviced again and again, and thus never leave the system. The next arrival of a job depends on when the user is finished. The difference between these to modelling paradigms is how they are parameterised. An open queuing network is parameterised by arrival rate, while a closed queuing network is parameterised by user think time and number of users in the system.

We are modelling an open system where users arrive at random moments, and they leave the system when the session is over. But to be able to perform the laboratory measurements with the available software, a fixed number of load-generators are used to represent a fixed amount of concurrent user sessions. Having a limited number of users also provide a consistent basis of comparison between measurements.

The laboratory setup has the properties of a closed queuing network. If the system is very busy, the arrival rate will decrease, as new customer cannot arrive until the current customer is finished. This

is a load generating limitation. But since the total response time (0.2seconds) is much smaller than the total average user session think time (515 seconds), it is possible to stretch the definition of an open queuing network.

Ideally, we would use a closed system to solve the model. Unfortunately, closed models only work when the workmix consist of only one class. That is, a user session is parameterised as one operation instead of 8 operations as in the BankApp example. When using a multiclass workmix, the class(operation) with the shortest service demand gets serviced more than the the other operations. If "Init" has half the service demand of "login", Init will be run twice as many times as login. This is not what we want: The workmix is supposed to specify the relative frequency between the operations. Bottom line, the closed model can be used only when all operations are combined into one operation.

When the system is higly utilised, and response time increases from tens and hundred of milliseconds to seconds, the arrival rate will be affected. The script waits for a request to complete before waiting for the next request, so the arrival rate and hence the load will decrease.

14.3 Closed single class queuing network

The spreadsheet for multiclass models was used to implement the single class model, by using only one class. The garbage colletor function determines the GC service demand as a function of the heap size. The GC service demands are added to the baseline parameters, and the sum is divided by the scale factor.

Queue 1 is the Web server, queue 2 is Application server, queue 3 is the Database server, and queue 4 is the network interface. Queue 5 represents the total session think time as a dependent queue.

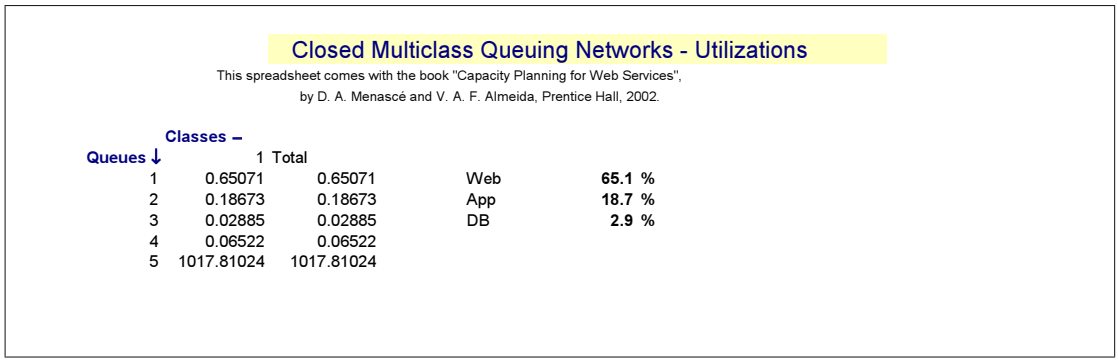
Closed Multiclass Queuing Networks

This spreadsheet comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business", by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

No. Queues:	5		**Tolerance:**	0.0005			
No. of Classes:	1						
	Classes –						
No. Requests per Class:	1020.00 *load*						
Throughput per Class:	0.0019763						
	Service Demand Matrix						
	Classes –					*Heap*	350
Type ↓							
Queues ↓ (LI/D/MPn)	1		projected	scalefactor	Baseline pa		GC
1 LI	329.25	<----------	329.25	1	320.00		9.25
2 LI	94.49	<----------	94.49	1	92.60		1.89
3 LI	14.60	<----------	14.60	1	14.60		
4 LI	*33.00*						
5 D	*515000.00*						
No. Iterations	Error						
27	0.00034776						

Figure 14.1: Dynamic model spreadsheet

The model is executed by pressing the "SOLVE" button which invokes a visual basic script to calculate the results. The button is not shown on the screenshot. The utilisations results are shown below:



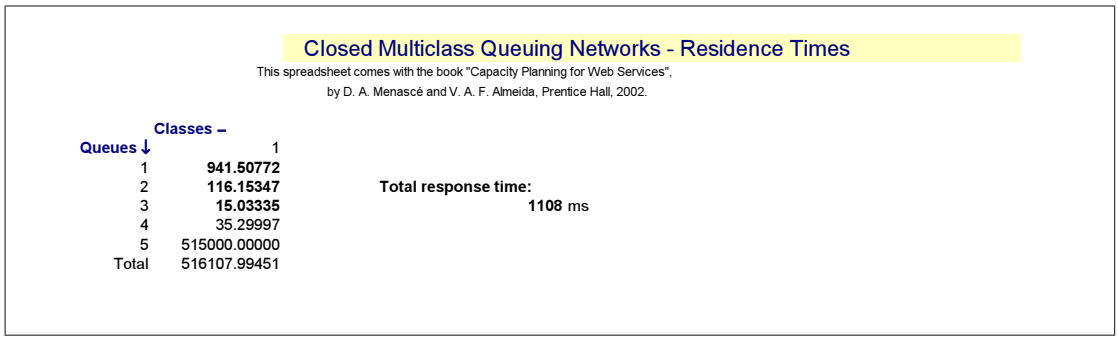
Closed Multiclass Queuing Networks - Utilizations

This spreadsheet comes with the book "Capacity Planning for Web Services",
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002.

Queues ↓	Classes – 1	Total		
1	0.65071	0.65071	Web	**65.1** %
2	0.18673	0.18673	App	**18.7** %
3	0.02885	0.02885	DB	**2.9** %
4	0.06522	0.06522		
5	1017.81024	1017.81024		

Figure 14.2: Dynamic model results

The total response time is 1079ms



Closed Multiclass Queuing Networks - Residence Times

This spreadsheet comes with the book "Capacity Planning for Web Services",
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002.

Queues ↓	Classes – 1		
1	**941.50772**		
2	**116.15347**	**Total response time:**	
3	**15.03335**	**1108** ms	
4	35.29997		
5	515000.00000		
Total	516107.99451		

Figure 14.3: Dynamic model results

14.4 Open multiclass queuing network

We compare the closed single class queuing network with an open multiclass network. The classes is given in the workmix definition in Section 8.3.

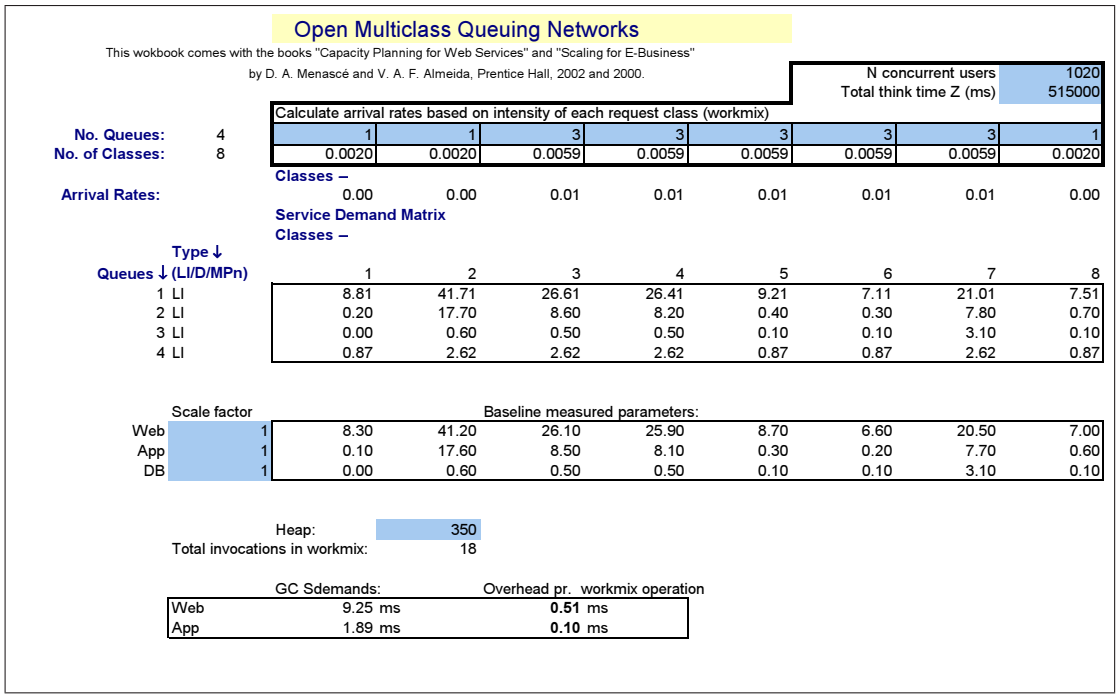
Each queuing centre in the dynamic model is parameterised with service demands from the static model, and the load is defined as the average number of concurrent user sessions in the system.

The GC service demands are calculated at the bottom of the spreadsheet. The service demands are divided by the scalefactor. This enables us to use the spreadsheet to project to a upgraded system. The total service demand is divided on all 18 operations in the workmix.

The baseline parameters are those obtained from single user parameter meaurements. We add the GC overhead to each web and ap server to the measured parameters, dividing by the scalefactor, and put the results in the model.

The network response time of 33ms is divided on the 8 operations.

The queues are the same as for the closed model, except that open models do not need a queue for the total think time. There are eight classes in the spreadsheet, where class 1 is init, class 2 is login, and so forth.



Open Multiclass Queuing Networks

This wokbook comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business"
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

N concurrent users 1020
Total think time Z (ms) 515000

No. Queues: 4
No. of Classes: 8

Calculate arrival rates based on intensity of each request class (workmix)

	1	1	3	3	3	3	3	1
	0.0020	0.0020	0.0059	0.0059	0.0059	0.0059	0.0059	0.0020

Arrival Rates: Classes –

	1	2	3	4	5	6	7	8
	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.00

Service Demand Matrix

Type ↓ / Queues ↓ (LI/D/MPn) — Classes –

		1	2	3	4	5	6	7	8
1	LI	8.81	41.71	26.61	26.41	9.21	7.11	21.01	7.51
2	LI	0.20	17.70	8.60	8.20	0.40	0.30	7.80	0.70
3	LI	0.00	0.60	0.50	0.50	0.10	0.10	3.10	0.10
4	LI	0.87	2.62	2.62	2.62	0.87	0.87	2.62	0.87

Baseline measured parameters:

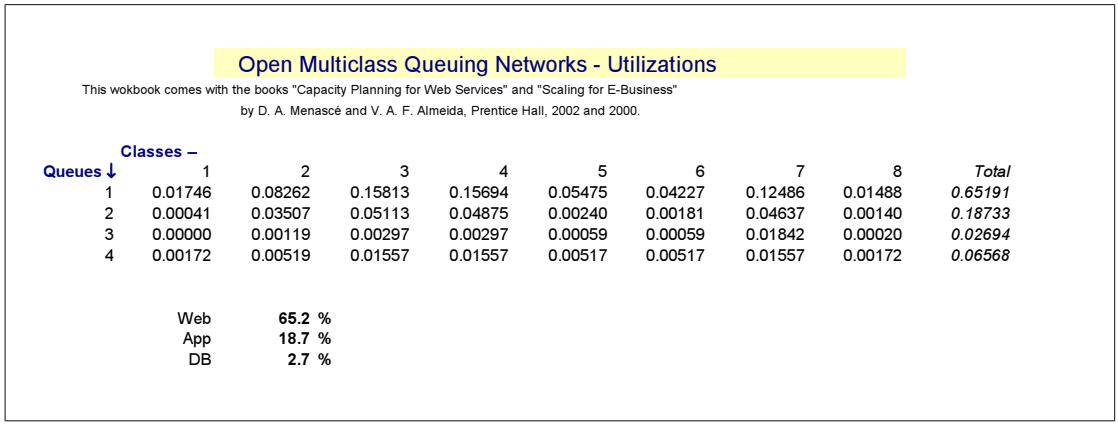
	Scale factor							
Web	1	8.30	41.20	26.10	25.90	8.70	6.60	20.50	7.00
App	1	0.10	17.60	8.50	8.10	0.30	0.20	7.70	0.60
DB	1	0.00	0.60	0.50	0.50	0.10	0.10	3.10	0.10

Heap: 350
Total invocations in workmix: 18

	GC Sdemands:	Overhead pr. workmix operation
Web	9.25 ms	**0.51** ms
App	1.89 ms	**0.10** ms

Figure 14.4: Dynamic model spreadsheet

The output from the solved model. Utilisation for each queue are specified, and the total utilisation for each hardware device.



Open Multiclass Queuing Networks - Utilizations

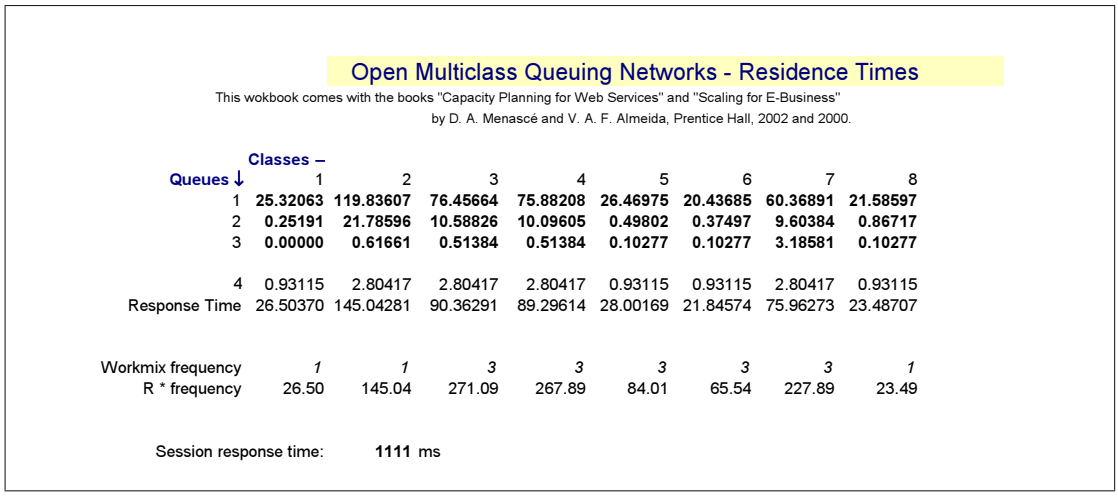
This wokbook comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business"
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

Classes –

Queues ↓	1	2	3	4	5	6	7	8	*Total*
1	0.01746	0.08262	0.15813	0.15694	0.05475	0.04227	0.12486	0.01488	*0.65191*
2	0.00041	0.03507	0.05113	0.04875	0.00240	0.00181	0.04637	0.00140	*0.18733*
3	0.00000	0.00119	0.00297	0.00297	0.00059	0.00059	0.01842	0.00020	*0.02694*
4	0.00172	0.00519	0.01557	0.01557	0.00517	0.00517	0.01557	0.00172	*0.06568*

Web	**65.2** %
App	**18.7** %
DB	**2.7** %

Figure 14.5: Dynamic model results



Open Multiclass Queuing Networks - Residence Times

This wokbook comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business"
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

Queues ↓ / Classes →	1	2	3	4	5	6	7	8
1	**25.32063**	**119.83607**	**76.45664**	**75.88208**	**26.46975**	**20.43685**	**60.36891**	**21.58597**
2	**0.25191**	**21.78596**	**10.58826**	**10.09605**	**0.49802**	**0.37497**	**9.60384**	**0.86717**
3	**0.00000**	**0.61661**	**0.51384**	**0.51384**	**0.10277**	**0.10277**	**3.18581**	**0.10277**
4	0.93115	2.80417	2.80417	2.80417	0.93115	0.93115	2.80417	0.93115
Response Time	26.50370	145.04281	90.36291	89.29614	28.00169	21.84574	75.96273	23.48707
Workmix frequency	*1*	*1*	*3*	*3*	*3*	*3*	*3*	*1*
R * frequency	26.50	145.04	271.09	267.89	84.01	65.54	227.89	23.49

Session response time: **1111** ms

Figure 14.6: Dynamic model results

The conclusion is that both open and closed models can be used in the range of BankApp experiment. If detailed info about each operation in the workmix is needed, then a open queuing network model can be used.

14.5 Alternative model: simulation

Just to show the possibility, the baseline is simulated by creating a simulation program in Java. The program code are presented in Appendix J. The simulation yielded the same utilisation levels as the analytical models. Figure 14.7 show the results from the simulation. The workload in this simulation is not step-wise, ie. that there is 8 sources that generates new requests simultaneously. The inter arrival time (IAT) for each request is calculated as

IAT = totalThinkTime / (intensity * numberUserSessions)

Total think time is 515 seconds, and intensity is the workmix definition for the request. For login the intensity is 1, and for newPayments the intensity is 3. Number of user sessions denotes the load.

We also conducted a simulation of a stepwise workload, where we arranged one source that put users in the queuing network. Each user then performed the workmix in sequential order and then left the system. The results were approximately identical to the results found in this section.

Title	*Order*	*pass*	*(Re)set*	*Users*	*Limit*	*Min*	*Now*	*Usage[%]*	*avg.Wait*	*QLimit*	*QMaxL*	*refus.*	*DL*
WebCpu	FIFO	no	0.0	69977	1	0	0	**64.75**	0.04329	unlimit.	26	0	no
AppCpu	FIFO	no	0.0	69977	1	0	1	**18.02**	0.00267	unlimit.	12	0	no
DatabaseCpu	FIFO	no	0.0	69977	1	0	1	**2.65**	2.0E-4	unlimit.	6	0	no

Figure 14.7: Results from BankApp simulation

14.6 Baseline validation

Before the model can be used to predict performance, we must validate that the model produces good estimates that is comparable to measurements under load.

14.6.1 Operating point iteration

The iterative process of reaching the operating point by measurements is somewhat simplified in this project. Instead of performing measurements only for operating point purposes, we use the results from the pilot measurements to find the load for the chosen operation point. From Figure 10.2 we see that the chosen operating point of 65% yields a capacity of N=1000 user sessions. We then perform one extra measurement to check that the load keeps the system at the operation point.

14.6.2 Comparing response time

Grinder reported the mean transaction times per operation in milliseconds. The transaction time is the elapsed time from a request is sent and till the whole response is received. Table 14.1 shows the total response time when adding the individual response times for each request reported by Grinder.

Request	Workmix freq.	Grinder response time	Session total
Init	1	33	33
Login	1	161	161
Payment	3	104	312
PaymentDetails	3	106	318
NewPayment	3	37	111
ConfirmPayment	3	34	102
PaymentReceipt	3	93	279
Logout	1	32	32
			1348

Table 14.1: Session response time reported by Grinder

The total session response time reported by Grinder is larger than the response time calculated in the dynamic model.

14.6.3 Baseline validation results

The dynamic model is validated with actual measurements. The results are presented in Table 14.2.

The load on both the dynamic model and baseline measurements were adjusted to reach the operating point, defined as 65% utilisation on the bottleneck device (web server). The number of concurrent user sessions at that utilisation level describes the capacity of the system. The model is now validated.

	Capacity: Number of concurrent user sessions
Model estimates	1020
Validation measurements	1000

Table 14.2: Baseline validation results

15 Projecting the upgraded model

In this chapter we project an upgraded model to the baseline. FromFigure 4.2 we remember that we needed a scale factor as input to the modification analysis. The first step is to determine the scale factor of the upgraded system. We then perform a modification analysis to project the upgraded model. Finally we validate the model by measuring the system.

15.1 Scale factor investigation

Wwe need to quantify a factor of the increased capacity of the upgraded system. Ideally we want strict and uniform scaling. Usually it is almost impossible to achieve a consistent scale-factor for all devices. To complicate matters, for example doubling the CPU clock frequency does not guarantee a doubling of the capacity of the device, since the instruction sets and pipelining properties may have been improved.

15.2 CPU comparison

A way to determine the size of the Pentium nodes in relation to the AMD nodes must be found. Table 15.1 shows a comparison of the CPU specifications.

The clock speed of the Intel CPUs cannot be directly compared to the AMD 1.4GHz 1600+ when evaluating their processing powers. AMD CPUs have a rating system, stating that an XP1600+ CPU can be compared to an Pentium 3 with a speed of 1600MHz.

Property/CPU	AMD Athlon 1600+	Intel Pentium4
Model name	Palomino	Northwood?
Clock speed (MHz)	1393	3400
Front Side Bus (MHz)	266 (2x133)	800 (2x400)?
L1 cache (Kb)	64	64
L2 cache (Kb) [1]	256	512
Bogomips [2]	2778,72	6789,52

Table 15.1: CPU specification comparison

15.3 Possible scaling factors

How close to uniform scaling are we? An evaluation of hardware properties follows for each of the three dimensions:

Processing:

- CPU clock speed is increased from 1400 to 3400MHz = factor of 2.4

- AMD rating system of a 1400MHz CPU (1600+), from 1600 to 3400MHz = factor of 2.1
- Memory bandwidth is increased from 266 to 800MHz = factor of 3.0
- L2 cache is increased from 256 to 512Kb = factor of 2.0

Storage:

- Memory size is unchanged from 1GB = factor of 1
- Java heap size is changed from 350 to 700 Mb = factor of 2
- Application footprint may be assumed to be constant. Assuming $ax+b$, where b is minimum footprint for a running server, and a is the increase of total memory and thread resources needed for more concurrent users(x). We would expect that the relationship between objects representing users in the system and available space will change for an scale-up.

Communication:

- Network bandwidth is changed from 100 to 1000 Mbit/s = factor of 10
- Message lengths for requests are assumed to remain identical = factor of 1

Conclusion: Since processing is the limiting factor, found by measurements and analysis, we assume that the processing dimension determine the overall scale factor.

- AMD rating system rates their CPU's higher that the clock speed, to compare them to Pentium3 or early versions of Pentium4 in terms of what these CPU's actually performed in benchmarks (for a given domain). This rating system is old and we do not expect it to be valid today. P4 design is more clever, so the CPU may perform more work pr. instruction cycle than earlier designs.
- Memory bandwidth is not the bottleneck.
- L2 cache should not have any impact, as the application software do not perform complex matrix calculations that could benefit from increased cache.
- CPU clock speed could be the determining factor since there should be some coarse relationship between the work done pr. instruction cycle.

Transaction systems are not about complex mathematical functions or floating point calculations, but rather pushing data back and forth, so performance could in our case be more about the CPUs ability to push data around and to do integer operations. We expect the scale factor to be between 2.1 and 2.4.

15.4 Measuring the scale factor

We want to establish a separate scale factor for web, application and database server. Since the database server is dependent to a certain degree on the hard disks, the database scale factor is expected to be lower than for the web and application server.

Single user stepwise measurements were performend to determine the resource usage on the upgraded system. See Section 13.2 for a description of how to perform those measurements. We also checked the effect of increased runlength, by changing the runlength of each test from 10 000 to 30 000.

Table 15.2 shows that the total service demand for a session is 194.5ms when running each test 10 000 times, while Table 15.3 shows that the service demand decreased to 180.5ms when running the test

30 000 times. We are not sure of the reason behind this decrease. The system should be steady after a few thousand invocations.

Operation	Web	App	DB	Web+App+Db SD	Workmix	Session SD
Init	4.4	0.0	0.0	4.4	1	4.4
Login	18.0	7.8	0.4	26.1	1	26.1
Payment	11.8	3.6	0.4	15.8	3	47.3
PaymentDetails	11.9	4.0	0.4	16.2	3	48.7
NewPayment	3.9	0.1	0.0	4.0	3	12.1
ConfirmPayment	2.5	0.0	0.1	2.6	3	7.6
PaymentReceipt	8.8	3.4	2.9	15.2	3	45.5
Logout	2.8	0.1	0.0	2.9	1	2.9
Total						194.5

Table 15.2: Upgraded nodes: Service demands for single user measurements. Runlength is 10 000.

Operation	Web	App	DB	Web+App+Db SD	Workmix	Session SD
Init	3.1	0.1	0.0	3.17	1	3.2
Login	17.7	6.7	0.4	24.8	1	24.8
Payment	10.9	3.4	0.3	14.7	3	44.0
PaymentDetails	11.4	3.4	0.3	15.2	3	45.5
NewPayment	3.1	0.0	0.0	3.2	3	9.5
ConfirmPayment	2.4	0.0	0.0	2.4	3	7.3
PaymentReceipt	8.7	3.2	2.8	14.7	3	44.1
Logout	2.1	0.0	0.0	2.1	1	2.1
Total						180.4

Table 15.3: Upgraded nodes: Service demands for single user measurements. Runlength is 30 000.

We can obtain the resource demands for each server by multiplying each operation by the workmix. Table 15.4 shows the results that we are going to use in the scale factor analysis in the next section.

Runlength	Web	App	DB	Session total
10K	141.7	41.2	11.6	194.5
30K	132.2	37.2	10.9	180.4

Table 15.4: Session resource demands for each server

15.5 Determine the scale factor

Two approaches are tried in finding the scale factor. First we benchmarked the baseline and upgrade nodes with single user measurements, and obtained a scale factor for the system as a whole. Then we tried to separate the scale factors for web, app and database server by performing single user measurements on the upgraded nodes to obtain the top level operation parameters.

The first scale factor: 2.5 Initially the scale factor was set by benchmarking the old and new nodes with single user measurements. Each user performed all operations in one session, and think

times were set to zero. 5000 user sessions were completed with one Grinder agent and one thread, to avoid contention. The number of completed sessions divided by the time it took gave the throughput. These measurements indicated a factor close to 2.5. Combined with the fact that measurements showed a CPU utilisation of 65% for both a load of N=1000 on the baseline and load N=2500 on the upgraded system. The capacity factor would then be 2.5.

Measuring a new scale factor: 2.3 A separate scale factor is established for each server by performing operation parameter measurements on the upgraded nodes:

		Service demands (milliseconds)			
	Runlength	Web	App	Database	Session total
Baseline	10K	320.0	93.6	14.6	428
Upgrade	10K	141.7	41.2	11.6	193
Ratio		2.25	2.27	1.30	2.2

Table 15.5: New scalefactor

One problem was discovered with the new scale factor: The measured baseline parameters adds up, but the upgrade parameters does not add up that well.

Baseline: Performing multiple user measurements, the session service demand was 436ms . Subracting the garbage collector service demands on the web and app server (8.1 + 3.7ms) leaves a total of 424ms. This is very close to the session total of 426ms found from single user measurements in Table 13.1.

Ugrade: The measured session service demand was 184.5ms, see Section 11.5. Subtracting GC service demands on web and app (3.1 + 0.7) yields a service demand of 180ms. But the parameter measurements showed a total of 194.5ms. We do not believe that a system with high load performes better than a system without much contention, since the baseline case in the previous paragraph indicates that we should get comparable results.

The final scale factor: 2.4 We decided to perform new operation parameter measurements on the upgrade, but instead of 10K sequential sessions as for the baseline, we now ran 30K. The execution time will now be approximately the same as for the baseline parameter measurements. It may have an effect that the servers in the baseline and the upgraded case run for an equally long time. The single user stepwise measurements result of 180ms is very close to the measured session service demand from multiple user measurements results, 180.7ms (184ms - 3.8ms of garbage collection)

		Service demands (milliseconds)			
	Runlength	Web	App	Database	Session total
Baseline	10K	320.0	93.6	14.6	428
Upgrade	30K	132.2	37.2	10.9	180
Ratio		2.42	2.51	1.34	2.38

Figure 15.1: Comparing service demands for baseline and upgraded

Conclusion: We have establised a scale factor for each server. The upgraded web, app and database servers have a scale factor of 2.4, 2.5 and 1.3 respectively.

15.6 Modification analysis

We must modify the baseline model to represent the projection. Figure 15.2 show a modified version of the open multiclass spreadsheet used in Figure 14.4.

Open Multiclass Queuing Networks

This wokbook comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business"
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

N concurrent users 2490
Total think time Z (ms) 515000

		Calculate arrival rates based on intensity of each request class (workmix)							
No. Queues:	4	1	1	3	3	3	3	3	1
No. of Classes:	8	0.0048	0.0048	0.0145	0.0145	0.0145	0.0145	0.0145	0.0048
		Classes →							
Arrival Rates:		0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.00

Service Demand Matrix
Classes →

Type ↓									
Queues ↓ (LI/D/MPn)		1	2	3	4	5	6	7	8
1	LI	3.52	17.23	10.94	10.86	3.69	2.82	8.61	2.98
2	LI	0.05	7.05	3.41	3.25	0.13	0.09	3.09	0.25
3	LI	0.00	0.46	0.38	0.38	0.08	0.08	2.38	0.08
4	LI	0.87	2.62	2.62	2.62	0.87	0.87	2.62	0.87

	Scale factor	Baseline measured parameters:							
Web	2.4	8.30	41.20	26.10	25.90	8.70	6.60	20.50	7.00
App	2.5	0.10	17.60	8.50	8.10	0.30	0.20	7.70	0.60
DB	1.3	0.00	0.60	0.50	0.50	0.10	0.10	3.10	0.10

Heap: 700
Total invocations in workmix: 18

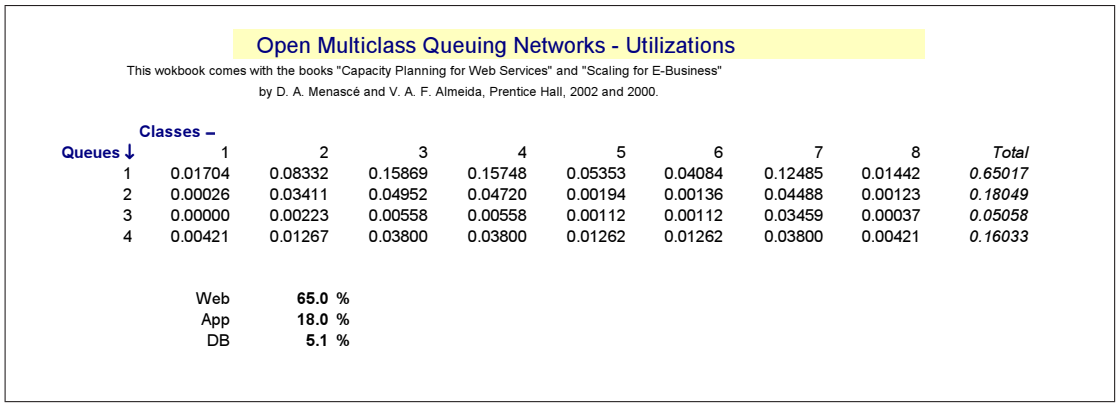
	GC Sdemands:	Overhead pr. workmix operation
Web	2.833 ms	**0.157** ms
App	0.628 ms	**0.035** ms

Figure 15.2: Dynamic model of upgraded system

These are the modifications we have done:

- The scale factors are altered to the values we found in the last chapter. (2.4, 2.5 and 1.3) We divide the scale factor with each measured baseline parameter. The result is the projected service demands in the service demand matrix.
- The heap size is set to 700MB. This results in reduced garbage collector service demands for web and app server. We then calculate the overhead per workmix operation. The GC service demands of 2.83ms and 1.16ms were divided by 18 workmix operations. This overhead is added to each web and app server operation in the service demand matrix.
- The load is adjusted until the model reaches the operating point of 65% web server CPU utilisation.
- The think time and workmix are not changed.

Figure 15.3 show the results from the modified baseline model. The web server CPU utilisation is 65% for a load of N=2490 concurrent user sessions.



Open Multiclass Queuing Networks - Utilizations

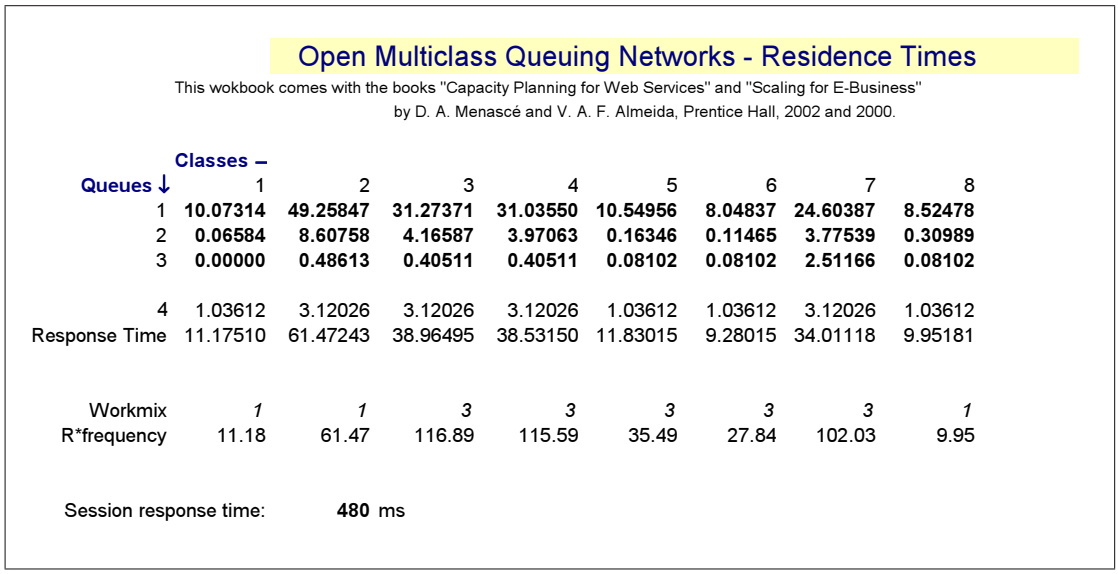
This wokbook comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business"
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

Queues ↓ / Classes →	1	2	3	4	5	6	7	8	*Total*
1	0.01704	0.08332	0.15869	0.15748	0.05353	0.04084	0.12485	0.01442	*0.65017*
2	0.00026	0.03411	0.04952	0.04720	0.00194	0.00136	0.04488	0.00123	*0.18049*
3	0.00000	0.00223	0.00558	0.00558	0.00112	0.00112	0.03459	0.00037	*0.05058*
4	0.00421	0.01267	0.03800	0.03800	0.01262	0.01262	0.03800	0.00421	*0.16033*

Web	**65.0** %
App	**18.0** %
DB	**5.1** %

Figure 15.3: Dynamic model of upgraded system: Utilisation results

In Figure 15.4 we see that the session response time was estimated to 489ms.



Open Multiclass Queuing Networks - Residence Times

This wokbook comes with the books "Capacity Planning for Web Services" and "Scaling for E-Business"
by D. A. Menascé and V. A. F. Almeida, Prentice Hall, 2002 and 2000.

Queues ↓ / Classes →	1	2	3	4	5	6	7	8
1	**10.07314**	**49.25847**	**31.27371**	**31.03550**	**10.54956**	**8.04837**	**24.60387**	**8.52478**
2	**0.06584**	**8.60758**	**4.16587**	**3.97063**	**0.16346**	**0.11465**	**3.77539**	**0.30989**
3	**0.00000**	**0.48613**	**0.40511**	**0.40511**	**0.08102**	**0.08102**	**2.51166**	**0.08102**
4	1.03612	3.12026	3.12026	3.12026	1.03612	1.03612	3.12026	1.03612
Response Time	11.17510	61.47243	38.96495	38.53150	11.83015	9.28015	34.01118	9.95181
Workmix	*1*	*1*	*3*	*3*	*3*	*3*	*3*	*1*
R*frequency	11.18	61.47	116.89	115.59	35.49	27.84	102.03	9.95

Session response time: **480** ms

Figure 15.4: Dynamic model of upgraded system: Response time results

15.7 Validation of projections

The performance estimates from the projected model must be validated with actual measurements on the upscaled system.

To find the capacity at the operating point, we use the same procedure as in Section 14.6. We evaluate the utilisation graph from pilot measurements to find the point where the utilisation is 65%. We then perform a measurement to check that the operating point is as accurate as possible. At operating point, the sustained load is N=2500 concurrent user sessions. The estimated capacity of the upgraded baseline is therefore N=2500

Table 15.6 shows that the estimates from the model is pretty close to measurement results. The projected model for the upgraded system is validated.

	Capacity: Number of concurrent users
Model estimates	2490
Validation measurements	2500

Table 15.6: Upgrade validation results

15.8 The gain of the upgraded system

The scale factor was found in Section 15.5. The capacity of the baseline and upgraded system were found when validating the baseline and upgraded model at operating point. The gain factor g(k) is found by dividing the measured capacity for upgrade by the baseline.

Baseline:

Size = 1

Capacity = 1

Upgrade:

Size = scalefactor = 2.4

Capacity = measured upgrade capacity / measured baseline capacity = 2500 / 1000 = 2.5

The gain in the processing dimension is 2.5 / 2.4 = 1.04. We have determined a slight superlinear scaling in the processing capacity dimension.

Part III

Feasibility

16 Experiences in applying SAM

In Part II we applied SAM to the case study. By applying SAM we have evaluated the method, and made some improvements to it along the way. This chapter presents the problems and challenges we met in the analysis. The technical problems that we experienced throughout the case study is documented in Appendix C, since we concentrate on the feasibility of SAM in this chapter.

16.1 Phase 1: Establishing the baseline

Establishing the baseline is pretty straightforward. Scaling objectives are determined and the workload is defined and implemented. The problems and challenges were more on the technical level, with installing the system and calibrating the test harness, but this is not directly related to the SAM method.

16.1.1 Implementing workload

When drawing users in a random fashion, sometimes users are drawn several times. The ratio of unique versus duplicate users should be low, and should ideally stay constant throughout all measurements. This means that each time we increase the number of concurrent users logged in to the system, the number of total users in the database should be increased accordingly. It is easy to check the ratio of unique users, since the measurement scripts compute these numbers automatically.

For this case study, measurements have shown that there is no caching of user data neither on the web servers nor the app servers. Experiments showed the same resource requirements whether a user was logged in over and over again, or that new users logged in each time.

16.2 Phase 2: Modelling the baseline

16.2.1 Pilot measurements

We found it necessary to perform pilot measurements as early as possible in the scalability analysis. The reason is that we rushed too fast about modelling in detail and planning measurements. Much of the work was never used later in the project.

The goal of pilot measurements is to establish the range of possible measurements, and also discover where any non-linear effects are by performing a search after non-linear effects. If possible it is also advised to perform measurements on an upscaled system as soon as possible. This gives a hint of the possible range of load that can be inflicted on the system, and may identify needed modifications to the test harness in order to be able to fully utilise the system.

If pilot measurements show that there may exist non-linear effects, one should start a search for possible sources of the non-linear effects. Various aspects of the system must be measured to find the source of the non-linearities.

16.2.2 Choosing the heap size

Choosing the heap sizes was not straightforward. The heap size has an impact on the garbage collector activity. We chose a baseline heap of 350MB and upraded heap of 700MB, since we at that time expected the scale factor between the upgraded system and the baseline to be approximately 2. We later determined the scale factor to be approximately 2.4, but kept the 2.0 scale factor of the heap. This introduced a skew from uniform scaling.

Another consideration is that with maximum load of N=3600 user sessions on the upgraded system, only 300MB out of the 700MB was used. This may be a low memory utilisation fraction in production systems.

16.2.3 Operating point

We found that pilot measurements also sets the ground for determining the operating point. Since we need the operating point throughout the analysis, this is a good thing. It is hard to formalise the process of determining the operating point. We suggested some rule of thumbs, and pointed out what factors we used in our analysis. We chose an equivalent operating point by reviewing the graphs of CPU utilisation and response times.

Because the resource requirements are not balanced on the servers, we chose to set the operating point on the bottleneck device. The bottleneck device was the web server with 65% CPU utilisation at operating point. The application server showed a utilisation of 19%, while the database server was at 3%. This is not optimal when analysing the J2EE architecture, which is implemented as the application server. The operating point should be defined for the application server instead. This can be accomplished with the further work suggested in Section 16.4.3.

16.2.4 Search for non-linear effects

In Chapter 11 we show how to search for possible non-linear effects.

We have not established if there is a direct connection between load dependence and non-linear effects, so this topic requires further work. We investigate load dependence on the baseline and upgraded system by plotting graphs of the CPU service demands for each server. In the memory dimension we search for effects in the paging routines, and for the connectivity dimension we show how to investigate application server connection pooling.

We have not looked into searching for software component level effects. This topic may require the use of a Java profiler. There are challenges attached to searching by profiling. Non-linear effects requires a high load to be measurable. We must also compare results on both the baseline and the upgraded system. Since the profiler adds overhead to the system, the overhead must be accounted for when measuring the system under load. This topic is considered further work.

The results from the search for non-linear effects are presented here:

- We found no significant load dependent effects on the baseline and upgraded system as a whole. We found a possible superlinear effect for garbage collector, but since the effect was only milliseconds out of a 430 millisecond user session total, we must use the result with caution.
- We noticed a steep increase in GC activity when the system reached CPU utilisation of 90% on both baseline and upgraded system, but we assume that the source of this is congestion effects. Such effects is well known and belongs to performance analysis and not scalability analysis according to our definition.

- Paging was non-existent in the range of all our measurements. The nodes have plenty of memory and can easily hold the heap sizes defined in this project. The heaps are not utilised more than one third to one half of the total heap size.
- The web and application servers do not use disk. The program files exist on NFS (network file system), and once they are started up the application only uses memory or access LAN. Until the a scenario where memory is close to full and excessive paging starts, we can ignore disk activity. OS background activities uses disk, but we assume these to be linear effects.
- It was not easy to obtain resource usage parameters for connection pooling. This must be investigated further. We were able to obtain the mean number of active connections to be approximately 3 for the upgraded system at full load of N=3600.

16.2.5 SP modelling

SP modelling presented many challenges. In Chapter 12 we summarise the experiences from working with SP modelling. In Appendix G we briefly present the work on transforming a complex model to a very simple and aggregated version.

We ended up with omitting the SP model by directly put top level operation parameters into the dynamic model.

16.2.6 Getting model parameters

Model parameters can be obtained either by analysis, component level measurements or system level measurements.

Fagerlie-Landmark[9] did much work on obtaining parameters by analysis and some specialised measurements. We were not able to repeat the analysis and the measurements because of lack of documentation. We simplified the modelling by omitting the component based model (SP).

One solution to obtain component parameters is Java profiling. The Java profilers do add an overhead to the running application, so results must be used with care. The ideal case would be to use low-overhead kernel profilers, but they do not give us information on the Java classes since they run in a Java Virtual Machine.

16.2.7 Dynamic modelling

We have shown that we could use both *open multiclass* or *closed singleclass* implementations of the queuing network with some restrictions. We have also shown how to implement an simulation model of the system in Java.

We tested both solutions by implementing the dynamic model as both a closed single class and a open multiclass queuing network. Finally we implemented the dynamic model as a simulation to check whether that also produced credible results. The result was both the closed model, open model and the simulation produced the same performance estimates. As long as the total think time is much greater than the total user session service demand, closed and open models can be used interchangeably.

In working with the dynamic models we ran into a few challenges about how to model the system. The main discussion was the choice of open or a closed model. Ideally we would want to simulate a real workload that is bursty and memoryless. But to be able to quantify effects we need controllable experiments, and in any way it is hard to simulate real world workloads in a laboratory setup.

A complicating matter is that we cannot use a closed queuing model for multiclass workloads, the problem is described in Chapter 14. A special case where we can use the multiclass workload is when the relative frequencies of all classes are equal, for example [1 login, 1 new payment, 1 logout].

16.3 Phase 3: Scalability of the upgraded system

The scale factor required quite a bit of work to be determined.

16.3.1 Scale factor analysis

The scale factor required quite a bit of work. Many iterations were needed to investigate the subject, both with various measurements and executions of the models for baseline and the upgraded system. In Section 15.5 we show that we tried to benchmark with single user measurements, using the whole user session as the work unit, but this proved insufficient. We had to separate the scale factors for the three servers.

We found that the the scale factor was CPU dependent, and it seems that the raw clock frequency increase factor of 2.4 maps quite well to the measured processing capacity of 2.4. We assumed initially by analytical reasoning that the CPU would be the limiting device, and that the scale factor would lie between 2.1-2.4.

16.4 Further work

Here we list some suggestions to further work. Some work is already performed on some of the subjects.

16.4.1 Improving toolbox

A lot of time went into trying to design the toolbox to make it as general as possible. But since it was mostly developed from the ground up and since we had little experience on the subject, there are some issues with the toolbox. These issues are good candidates for further work, since they would improve the toolbox, and automate the process further:

- The toolbox currently instructs Sysstat to output the CPU utilisation and paging to separate log files. Sysstat enables output to binary files instead of text output. This way many parameters can be logged into one file. The toolbox scripts involved in analysing and viewing logs must be edited to facilitate such a new feature.
- The results from the measurement should be recorded in for example a text file, with one line for each record. This would make it easier to import the results to spreadsheets, and make automatic calculation easier. The format could be:

  Time, Load, Heap, DB: X, R(session), Web1: U, gc, avgHeap, maxHeap , APP1: U, gc, avgHeap, maxHeap, DB1: U, R, Test1 Test2 ...

16.4.2 Adding RMI component to SP

We suggest to add an RMI component to the SP model if using a separate web and application server configuration as we do in this project.

Why:

> As shown in Appendix D, the network messaging overhead is pretty huge when using a separate web and application server. It may be worth separating out RMI as a SP component.

How:

> How to obtain resource parameters for RMI and network messaging requires work. We have shown how to measure the resource usage for the workmix operations and compared them to the baseline, but we do not know if these effects can be tributed to RMI and networking messaging alone. This requires further investigation.

16.4.3 Isolating application server

The application server layer could be "isolated" the by ignoring the web server layer. One can then investigate the interaction between application and database layer. With the current baseline, a lot of work is required to generate enough load from the web servers to fully utilise the application server.

Why:

> J2EE architecture is the focus. Since it is the application server that effectively implements J2EE architecture, the operating point for the baseline system should be defined for the application server rather than the web server. The current situation is that operating point is defined for the web server since it is the bottleneck.
>
> It can be argued that web nodes are fully scalable under certain conditions: They can be replicated to balance the load between them. As long as a session is terminated and not cached, any node should be able to handle a subset of the logged in users without any scalability issues. If session fail-over is a requirement, for example two and two web nodes can be paired to replicate each other's states. This would introduce some overhead, but will presumably not introduce big non-linear effects. If the state were to be replicated to all web servers, this would more likely introduce non-linear effects.

How:

> Grinder load generator can emulate the interaction between the web server and application server. EJB requests can be issued directly from Grinder to JBoss

Status:

> Work has been done to implement the direct invocation of EJB services. Measurements showed only 1/5 of the CPU utilisation for the (assumably) same load. There are two possibilities: either an error in the test harness, or it may be that the web server uses the application server in a way unknown to us. The JNDI lookups are not expected to be so resource intensive that they can account for the differences. The source code for the load scripts are found in Appendix E.

16.4.4 Obtain parameters with Java profiler

Why:

If a system requires more detail in the SP model that the SP model constructed in this project, components must be split in two or more components. Then the effects of each component has to be separated from the other. This requires a profiler.

But before the profiler output can be used, work must be done to reduce the profiler overhead to an acceptable level and/or quantify the overhead of the profiler so one can adjust the results afterwards.

Status:

Some work was performed in using profiling to obtain resource usage on a component level. But it was not easy to set up filters to obtain resource usage for groups of classes. It was also a problem in analysing and understanding how the profiler calculated its results. The problem arises when external libraries are used, or when the software uses IO or other external resources.

There exist a system level profiler for the Solaris operating system which can trace Java methods in a similar way to C functions. This removes the need for a resource-hog application as a professinal Java profiling tool is. [1]

16.4.5 Override grinder.sleepTimeVariation

Why:

Grinder only supports uniformly distributed sleep times. We have assumed that the random nature of the experiment is good enough for our purpose. Each Grinder node start with a delay, and the threads start in a non-deterministic way. However, we have not performed a statistical analysis to back up the assumption. This might be analysed as further work.

How:

In the Grinder workmix script (that generates the load), we simulate user think time by using grinder.sleep:

```
grinder.sleep(10000)
```

But this makes Grinder sleep with a fixed amount of milliseconds: So we have combined this with the sleepTimeVariation property in the Grinder properties file. The sleep times will be varied with a normal distribution.[2]

```
grinder.sleepTimeVariation=0.4
```

One possibility is to avoid Grinder's sleep time variation. We remove the property line (or comment it out) from the Grinder properties file.

```
#grinder.sleepTimeVariation=
```

[1] `http://www.sun.com/bigadmin/content/dtrace/`

[2] From the Grinder documentation: `http://grinder.sourceforge.net/g3/properties.html` *"The Grinder varies the sleep times specified in scripts according to a Normal distribution. This property specifies a fractional range within which nearly all (99.75%) of the times will lie. E.g., if the sleep time is specified as 1000 and the sleepTimeVariation is set to 0.1, then 99.75% of the actual sleep times will be between 900 and 1100 milliseconds."*

Instead, we can define our own distribution in the Grinder workmix script. We can set up a distribution based on the RngPack package *http://www.honeylocust.com/RngPack/* This package offers better random generators than Sun Java's Math.Random. Uniformly distributed numbers from RngPack can be transformed to negative exponential distributed numbers, and then be fed to *grinder.sleep*:

```
grinder.sleep(NegExpDist.draw(10000))
```

17 Conclusion

This chapter summarizes the project. We list the key experiences obtained when performing a SAM analysis, and propose improvements to SAM. We then state the main contributions to the research and evaluate if the goals are achieved.

17.1 Summary

In this thesis we have applied the research methods presented in Section 1.4. We have presented the SAM method and applied it to the case study. Along the way we have evaluated SAM and improved it by proposing changes. The procedure of the analysis is documented in detail, and now we hope that research can be taken further as a result of this thesis.

A model was built of the BankApp application, and measurements were performed to obtain model parameters. The baseline model predicted the performance very well. We then projected the upgraded system by modifying the baseline model, and this model also predicted the performance well compared to the measurements.

17.2 Problems encountered

In Appendix C we list the problems and experiences encountered in the case study. A brief summary is presented here:

- The challenges of maintaining the toolbox for two separate clusters is described in the chapter, and it also describes problems that arose when the baseline nodes were taken offline, and baseline experiments could not be repeated.
- We found that the FL model was not properly calibrated, which emphasizes the need for controlled and accurate experiments.
- The problems and experiences with repeating the experiments from the Fagerlie-Landmark project are listed. The result of these experiences was the acknowledgement of the need for SAM infrastructure and better documentation.

17.3 General feasibility of SAM

In Chapter 16 we evaluated SAM for the case study. SAM proved to be feasible in the context of J2EE and the BankApp application.

We have shown how to perform all steps of a SAM scalability analysis. The main obstacle was component modelling and how to obtain the parameters for the model. We omitted component modelling and only used a system level model. We used a load dependent function in the spreadsheet that implemented the dynamic model. This is feasible if we only have a few functions. When one have a lot

of dependencies, a static model should be used as the component model. We have also shown how to implement the system level model as a simulation, if more detailed dynamic modelling is needed.

Our study was in a reserach context, where we had access to two sizes of a system which we could measure. We could obtain the parameters for the upgraded system on a system level basis. In a developer or capacity planning context one normally do not have access to an upgraded system, and then one has to use component modelling and modification analysis to estimate the performance of an upgraded system.

17.4 Suggestion for SAM improvement

In Section 4.3 we presented the main steps of the SAM procedure. The steps indicate that we first build a baseline model, measure its parameters and validate with baseline measurements. Then we project the upgraded model and finally validate the upgrade by measurements.

We suggest that pilot measurements are performed as early as possible, and upgrade measurements should be performed as well if possible. The SAM main steps are logical and stepwise on a intuitive level, but for applying SAM in practice, some advice to the measurement engineer can be given:

1. First install and configure the system. Extensive testing is probably required to eliminate errors and mis-configurations in the test harness.
2. Measure the upgraded system if possible. The upgraded system may not be available in a development context, but it may be available in an capacity planning or researcher context. We want to measure the upgraded system to discover any problems with the test harness. We experienced in this project that the test harness had to be modified to be able to measure with a high load on the upgraded system. Only a few system configuration details should separate the baseline from upgraded configuration, so it should be easy to perform upgraded measurements in the same batch.
3. Before investing precious time on detailed modelling, pilot measurements should be performed to get a feel for the possible range of measurements and what kind of upgraded system one wants to measure, and to check if there are any prominent load-dependent variables or non-linear effects. The reason is that a model should not be more complex that it has to be.

17.5 Main contribution

The main contribution to the scalability research lies in the measurement domain. The toolbox is a result of developing SAM infrastructure support. In this section we evaluate the toolbox and the documentation contribution of the project.

SAM walk through

This thesis shows how SAM can be performed and give credible results. We show how to apply SAM in Chapter 4, and how to perform the practical steps in the case study in Part II. SAM is evaluated in the context of the case study in Chapter 16.

Toolbox

To obtain credible measurements, a lot of work is involved. Since professional load generators are too expensive, open source software must be used. The professional tools offer integration with the

application, and automation in all aspects of designing measurement tests. Finally, these tools offer statistics analysis. All this has to be done manually with the tools available for us, and glue the tools together using scripts, and write small programs to analyse the data.

The toolbox provides a flexible way to implement measurement experiments on a Unix system. For new applications, the idea is to enable peer students to adapt the scripts as needed to get the job done. Even if one has to rewrite everything, the scripts contain valuable unix knowledge.

We list some of the key evaluations of the toolbox:

- The toolbox proved very useful for this project. Managing all measurements would have been very cumbersome without the automatisation. For example: In Figure 11.3, 30 measurements were required for that graph only, where each measurement required a modified configuration. Considering that each measurement took approximately 90 minutes, the toolbox was cleary invaluable. The toolbox enabled us to perform the task in 3 batches. This in turn could easily be run in one batch by adding another script to automate the three batches.
- The toolbox was to some extent proven easy to use. This is based on the the observation that configuring the system for new measurements (using load balancer and several web servers) took only a few hours, and it had been 5 months since the last time I had used the measurement setup.
- Rød-Mongstad[10] used many scripts from the toolbox for their project, and Ruud-Tveiten[11] used some Clustis2-related scripts. Ruud-Tveiten built their own resource function workbench, but based it on some of the toolbox scripts.
- The experiences from this project is that it is relatively easy to add custom configurations. It is a matter of editing the configuration and experiment files to suit the needs.

The toolbox was being developed at the time of Ruud-Tveiten and Rød-Mongstad projects. Since then the scripts got better, and documentation was written. In my opinion, the toolbox would have been more useful to those projects if they had the current version to work with.

Significant measurements

Here we list some of the significant measurements:

- **Steady state for garbage collection:** In Section 10.8 we showed that the garbage collector was not in steady state until 50 minutes out in the experiment. For thiscase study it only resulted in a few milliseconds extra service demand per user session, where the user session service demand was approximately 190 for the upgraded system.
- **Garbage collector dataload dependent function:** Section 13.3 presents how we obtained the garbage collector resource usage as a dataload dependent function. The service demand is a function of the heap size.
- **Data transfer network bandwidth requirement:** In Appendix F we measure and derive the total bandwidth requirement for a user session.
- **Scale factor of upgraded system:** Section 15.5 presents how we determined the scale factor, first by analytical approach, then obtain the scale factor by measurements.
- **RMI overhead:** We investigated the network messaging overhead of our baseline configuration by measuring an alternative configuration. The baseline configuration consist of a separate web and application server. In Appendix D we show measurements on an integrated configuration where the web server is integrated in the application server. The message overhead resulted in approximately a doubled service demand for the separate configuration. This means that using one extra node in the separate configuration is equal to using only one node for the integrated configuration.

17.6 Goal achievement

We evaluate the criteria presented in Section 1.5, and find that they are met:

1. *"Describe in a detailed fashion how to apply SAM to a system, and capture the considerations behind the choices."*

   The report shows how to perform all steps in a scalability analysis using SAM. Chapter 5 shows how we apply all steps of SAM to the case study, and each chapter in Part II contains detailed information on why and how we have performed the steps.

2. *"Provide a toolbox. This enables peer students to focus on bringing the Scalability Assessment method one step further, rather than doing much of the same work all over again."*

   The toolbox is provided and is evaluated in the previous section.

3. *"Apply SAM to investigate key scalability issues for this case study. The focus is on the processing dimension, but storage and connectivity dimension must also be investigated."*

   - The main focus is on the processing dimension. This is the easiest and most intuitive dimension to work in. We have analysed the processing dimension throughout the the case study, and in particular in chapters 10, 11 and 13.
   - We partly studied the storage dimension in connection to memory management: We obtained a storage (heap) dependent function in Section 13.3. Paging was investigated in Section 11.9 and found to be virtually non existent on the web and application servers in this case study.
   - Finally the connectivity dimension is looked at when analysing connection pooling in Section 11.10. Threading constraints in the JVM limited the maximum number of user sessions on the the web servers. This was an issue when using persistent HTTP connections. See Section 6.5.3.

Bibliography

[1] HUGHES, P. H. *Assessing Scalability.* Working draft, IDI, NTNU, May 2002

[2] HUGHES, P. H. *The Scalability Assessment Method: Outline Procedure for a Prototype SAM Engine.* Working draft, IDI, NTNU, March 2006

[3] HUGHES, P. H. *Towards a Theory of Scalability for Computing Systems*. Working draft, IDI, NTNU, March 2005

[4] HUGHES, P. H. *Lecture Notes in Performance Engineering TDT4220.* IDI, NTNU, 2005

[5] HUGHES, P. H. *Structure and Performance Specification*. Extract from "SP Principles". STC Technology Ltd, 1988

[6] HUGHES, P. H. *Considerations relating to SP model parameterisation*. Working paper IDI, NTNU, October 2004

[7] BRATAAS, G. AND HUGHES, P. H. *Exploring Architectural Scalability: A J2EE case study*. WOSP'04: Fourth International Workshop On Software and Performance. California, USA, January 14-16, 2004.

[8] HUGHES, P. H., BRATAAS G., FAGERLI, J.-A. AND LANDMARK, O. C. *Exploring the Scalability of an Enterprise Architecture*. Working paper IDI, NTNU

[9] FAGERLI, J.-A. AND LANDMARK, O. C. *Scalability of a Platform for financial services based on the J2EE platform.* Master's thesis IDI, NTNU. Trondheim, July 2003

[10] RØD, E. AND MONGSTAD, E. *Model-driven Measurement of a Bank System*. Master's Thesis, IDI, NTNU, June 2005.

[11] GISLE, O. AND RUUD, J. *Measuring on Large-Scale Read-Intensive Web Sites.* Master's Thesis, IDI, NTNU, June 2005

[12] HOLMEFJORD, A. *Model-based Framework for Scalability Assessment.* Master's Thesis, IDI, NTNU, June 2006.

[13] EDB BANK & FINANS A/S. *Transigo Documentation.*

[14] BARISH, G. *Building Scalable and High Performance Java Web Applications Using J2EE Technology*. Addison Wesley Professional, 2001

[15] MENASCÉ, D. A. AND ALMEIDA, V. A. F. *Capacity Planning for Web Services.* Prentice Hall, 2002

[16] LILJA, D. P. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press, 2000.

[17] GUPTA, A. AND DOYLE, M. *Turbo-charging Java HotSpot Virtual Machine, v1.4.x to Improve the Performance and Scalability of Application Servers*. SUN. JVM Garbage Collector `http://java.sun.com/developer/technicalArticles/Programming/turbo/`

[18] Clustis2 homepage `http://clustis2.idi.ntnu.no/`

[19] Jakarta Tomcat site `http://jakarta.apache.org/tomcat/`

[20] JBoss home page `http://www.jboss.com/`

[21] The Grinder home page `http://grinder.sourceforge.net/`

[22] Sysstat utilities home page http://perso.wanadoo.fr/sebastien.godard/

[23] Web site for "Capacity Planning for Web Services" file downloads http://cs.gmu.edu/~menasce/webservices/

[24] SUN Java garbage collector http://java.sun.com/docs/hotspot/gc1.4.2/

[25] WebLogic Cluster Architectures http://e-docs.bea.com/wls/docs81/cluster/planning.html

Part IV

Appendix

A Parameter calculation

INIT	
Total requests:	10000
Start	12:14:32
Stop	12:16:19
Diff	00:01:47
Seconds	107 s
Throughput:	93.46 tps
WEB	
Utilisation	77.90 %
Service demar	8.3 ms
APP	
Utilisation	0.81 %
Service demar	0.1 ms
DB	
Utilisation	0.21 %
Service demar	0.0 ms

Init :

Web	**8.3** ms
App	**0.1** ms
Db	**0.0** ms

Init+Login+NewPaym	
Total requests:	10000
Start	06:28:39
Stop	06:42:09
Diff	00:13:30
Seconds	810 s
Throughput:	12.35 tps
WEB	
Utilisation	71.95 %
Service demar	58.3 ms
APP	
Utilisation	22.25 %
Service demar	18.0 ms
DB	
Utilisation	0.91 %
Service demar	0.7 ms

NewPaym:

Web	**8.7** ms
App	**0.3** ms
Db	**0.1** ms

INIT+LOGIN	
Total requests:	10000
Start	05:02:36
Stop	05:14:14
Diff	00:11:38
Seconds	698 s
Throughput:	14.33 tps
WEB	
Utilisation	70.96 %
Service demar	49.5 ms
APP	
Utilisation	25.34 %
Service demar	17.7 ms
DB	
Utilisation	0.95 %
Service demar	0.7 ms

Login:

Web	**41.2** ms
App	**17.6** ms
Db	**0.6** ms

Init+Login+NewPaym+ConfPaym	
Total requests:	10000
Start	06:50:13
Stop	07:05:12
Diff	00:14:59
Seconds	899 s
Throughput:	11.12 tps
WEB	
Utilisation	72.15 %
Service demar	64.9 ms
APP	
Utilisation	20.23 %
Service demar	18.2 ms
DB	
Utilisation	0.94 %
Service demar	0.8 ms

ConfPaym:

Web	**6.6** ms
App	**0.2** ms
Db	**0.1** ms

INIT+LOGIN+PAYM	
Total requests:	10000
Start	05:19:27
Stop	05:37:07
Diff	00:17:40
Seconds	1060 s
Throughput:	9.43 tps
WEB	
Utilisation	71.37 %
Service demar	75.7 ms
APP	
Utilisation	24.70 %
Service demar	26.2 ms
DB	
Utilisation	1.09 %
Service demar	1.2 ms

Paym:

Web	**26.1** ms
App	**8.5** ms
Db	**0.5** ms

Init+Login+NewPaym+ConfPaym+PaymReceipt	
Total requests:	10000
Start	07:15:27
Stop	07:35:25
Diff	00:19:58
Seconds	1198 s
Throughput:	8.35 tps
WEB	
Utilisation	71.25 %
Service demar	85.4 ms
APP	
Utilisation	21.62 %
Service demar	25.9 ms
DB	
Utilisation	3.26 %
Service demar	3.9 ms

PaymRecpt:

Web	**20.5** ms
App	**7.7** ms
Db	**3.1** ms

INIT+LOGIN+PAYM+PAYMD	
Total requests:	10000
Start	05:56:01
Stop	06:19:38
Diff	00:23:37
Seconds	1417 s
Throughput:	7.06 tps
WEB	
Utilisation	71.65 %
Service demar	101.5 ms
APP	
Utilisation	24.16 %
Service demar	34.2 ms
DB	
Utilisation	1.17 %
Service demar	1.7 ms

Paym:

Web	**25.9** ms
App	**8.1** ms
Db	**0.5** ms

Init+Login+Logout	
Total requests:	10000
Start	11:19:23
Stop	11:32:48
Diff	00:13:25
Seconds	805 s
Throughput:	12.42 tps
WEB	
Utilisation	70.27 %
Service demar	56.6 ms
APP	
Utilisation	22.72 %
Service demar	18.3 ms
DB	
Utilisation	2.34 %
Service demar	1.9 ms

Logout

Web	**7.0** ms
App	**0.6** ms
Db	**1.2** ms

Figure A.1: Excel spreadsheet for calculating service demands

This spreadsheet shows how service demand parameters are calculated. The blue fields are input from measurements. The throughput is calculated for either single request (init) or aggregated requests (init+login, ...). For each request, the CPU utilisation on each node is measured. Dividing the utilisation for a device (the CPU in this case) on a device by the the throughput to obtain service demand for that aggregated request on the device. To obtain the service demand for a single request type, the demand for the previous group is subtracted.

The boxes shows the service demand for each individual request on each server.

B Measurement results

B.1 Connection pooling results

Connection pooling was analysed in Section 11.10. Here are the derived results from the MySQL log file. The upgraded system was measured with a load of N=3600 and heap=700MB.

The approach is to divide the queries in the MySQL log in partitions of 4000 queries in each . We then count the unique connection IDs for each partition. That will give us the number of active connections for each partition. We calculate the mean of all partitions, and obtain a mean for the overall connection pooling.

Listing B.1: output from analyse-mysql-logs.sh

```
Unique connections. pr. interval: 5       Active:  4 1 3 2 5
Unique connections. pr. interval: 3       Active:  1 3 2
Unique connections. pr. interval: 3       Active:  1 3 2
Unique connections. pr. interval: 4       Active:  4 1 3 2
Unique connections. pr. interval: 3       Active:  4 3 5
Unique connections. pr. interval: 2       Active:  4 5
Unique connections. pr. interval: 3       Active:  6 4 5
Unique connections. pr. interval: 2       Active:  6 5
Unique connections. pr. interval: 3       Active:  6 7 5
Unique connections. pr. interval: 5       Active:  8 6 7 9 5
Unique connections. pr. interval: 3       Active:  8 9 5
Unique connections. pr. interval: 2       Active:  9 5
Unique connections. pr. interval: 3       Active:  10 9 5
Unique connections. pr. interval: 2       Active:  9 5
Unique connections. pr. interval: 6       Active:  11 13 9 12 14 5
Unique connections. pr. interval: 3       Active:  13 12 14
Unique connections. pr. interval: 3       Active:  13 12 14
Unique connections. pr. interval: 2       Active:  13 14
Unique connections. pr. interval: 11       Active:  21 17 20 15 14 22 18 23 13 16 19
Unique connections. pr. interval: 3       Active:  21 22 23
Unique connections. pr. interval: 3       Active:  21 22 23
Unique connections. pr. interval: 4       Active:  21 22 24 23
Unique connections. pr. interval: 2       Active:  24 23
Unique connections. pr. interval: 3       Active:  25 24 23
Unique connections. pr. interval: 3       Active:  25 24 23
Unique connections. pr. interval: 3       Active:  25 24 23
Unique connections. pr. interval: 3       Active:  25 24 23
Unique connections. pr. interval: 3       Active:  25 24 23
Unique connections. pr. interval: 6       Active:  27 25 28 26 23 29
Unique connections. pr. interval: 2       Active:  27 29
Unique connections. pr. interval: 3       Active:  27 28 29
Unique connections. pr. interval: 3       Active:  27 28 29
Unique connections. pr. interval: 3       Active:  27 28 29
Unique connections. pr. interval: 3       Active:  27 30 29
Unique connections. pr. interval: 3       Active:  27 30 29
Unique connections. pr. interval: 3       Active:  27 30 29
Unique connections. pr. interval: 2       Active:  30 29
Unique connections. pr. interval: 3       Active:  30 31 29
Unique connections. pr. interval: 3       Active:  32 29 31
Unique connections. pr. interval: 3       Active:  32 29 31
Unique connections. pr. interval: 3       Active:  32 31 29
Unique connections. pr. interval: 2       Active:  32 29
Unique connections. pr. interval: 3       Active:  33 32 29
Unique connections. pr. interval: 3       Active:  32 33 29
Unique connections. pr. interval: 2       Active:  33 32
Unique connections. pr. interval: 3       Active:  33 32 34
Unique connections. pr. interval: 3       Active:  33 32 34
Unique connections. pr. interval: 3       Active:  33 32 34
Unique connections. pr. interval: 3       Active:  32 33 34
Unique connections. pr. interval: 2       Active:  33 34
Unique connections. pr. interval: 3       Active:  35 33 34
Unique connections. pr. interval: 3       Active:  35 33 34

Number of SQL queries pr. interval: 4000
Average number of connections:   3.2   Max: 11
```

C Problems encountered

In Section 17.2 we summarised the problems listed here.

This chapter points out the problems and technical issues that were encountered during the phases of installing and configuring the software application, calibrating and testing the software, to the measurement phase.

C.1 Two separate clusters

The first installation of BankApp were done on Clustis. This was a cluster of 38 computer nodes with AMD CPUs with a frequency of 1400 to 1600 MHz, and network bandwidth of 100Mbit. Several measurements were performed here, but is was hard to do any validation measurements without nodes with higher capacity. Luckily, NTNU bought new cluster called Clustis2, with Intel Pentium 4 nodes equipped with of 3400GHz and gigabit network interfaces.

Porting the measurement framework to the new cluster required some work: Application software had to be recompiled and reinstalled, and some software had to be reconfigured. Scripts had to be changed a bit to to work on the new cluster framework. The disk setup, the network file system and other software properties were different, and had to be taken into account. The two clusters existed totally separated from each other, so the scripts had to be synchronised manually between clusters. This was a cumbersome task, and several times the synchronising failed and lead to incompatible measurements.

Finally, a few nodes from the old Clustis were moved to the new Clustis2, so accurate measurements could be performed. Now the baseline and upgraded system could be compared in a heterogeneous environment, excluding operating system and platform differences from the analysis.

The old Clustis nodes were available a while until they were taken offline because they often needed extra care when the cluster software or kernel version were upgraded. In addition to that, the old Clustis became unusable since it required an ever increasing effort to keep it online. This made it impossible to repeat experiments on the baseline.

C.2 Problems identified by measurements

Since this project builds upon the work of Fagerlie-Landmark, we outline some problems found in their thesis. We point this out to illustrate how difficult and work-intensive the scalability analysis process is. It also acts as a rationale to explain why we perform work that over again that seemingly is already done.

The calculated service demands in the FL excel spreadsheet is based on estimation of CPU instruction count from a very complex and detailed SP model. If we compare them to parameter measurements from this project, we see that they do not have the same relative ratios. We look at some numbers for the web server in Figure C.1.

The table shows three top level operations on the system. Column two contains the instruction count estimated by FL [9], and column 4 contains the service demands measured in this project. These to metrics have different units, but both express the amount of work that the operations require from the CPU. If we normalise the numbers with respect to the "initialise" operation, we find the ratios between the operations. The FL estimations do not compare to the measurements. This point illustrates the need for calibrating the model by accurate measurements.

Operation	FL instruction count	FL ratio	Measured service demands	Measured ratio
Initialise	1,2632582E-02	1,0	8,7ms	1,0
Login	1,4834781E-02	1,2	40,6ms	4,7
Payments	1,9891225E-02	1,7	25,2ms	2,9

Figure C.1: Comparing FL devolved work to measured service demands

C.3 Problems with repeating FL measurements

These are early experiences that indicated the need for SAM infrastructure and for better documentation of the process:

- There existed only one copy of the experiment configuration files. Configuration changes were done on one instance of the configuration files. This way we could only find information about the last setup.
- The measurement script shown in the report did not reflect the actual workmix that was defined earlier in the FL report. Some sleep times were wrong, and one sleep call was wrongly placed. We did not find any "correct" measurement scripts in the home directory on the cluster either.
- There were no indication of resetting the database between each experiment, so we assume the database grew for each measurement run, since users add payments under experiments. Start conditions should be as equal as possible for each experiment.
- There were few details on what was actually done in the experiments and measurements. There were no records on what setup was used in each configuration.
- The method of how to obtain parameters for SP was only sketched briefly, and it was not enough to just go ahead and reproduce the results.

C.4 Workmix script user limitation

The original Grinder workmix script had limitations on how many customers that could be tested.

The Grinder workmix script needs a list of what customers to test. This information is extracted from the database. The script file contained all the customers and payment IDs to be tested. This limited the maximum number of customers to approximately 500-1000 because of Jython script array limitations. Using larger resulted in a runtime error. For a long run test, this was not enough to avoid the reuse of customers.

We had to rewrite the representation of users on the load generating nodes by reading customers randomly from a file. See Section H.1 for a code listing of the Grinder script.

We added the DataSource class to the workmix script and smaller modifications to be able to utilise the new class. The customer file was created in the process when the database was generated, see Section H.2.

C.5 Error in workmix script

The workmix script had an error, resulting in that Grinder logged in a user successfully, but the subsequent user requests were not "connected" correctly to a session. Therefore all requests after login were as if no user was logged in. This resulted in a low load on the servers, since web server only returned the main page instead of handling user transactions. We assume that the MySQL logs were not checked properly to catch this bug. These logs revealed that MySQL got requests for attribute values=null (no reference). Instead of "null" there should be a nine-digit number indicating the user ID. This problem was localised using a HTTP proxy sniffer that intercepted the communication between the Grinder and Tomcat.

This error resulted in a much lower utilisation on the Tomcat and especially JBoss server than it should be. Since the servers got requests from a user that was not logged in, they just returned a response with the login page. Naturally, this requires relatively little work from the servers. FL showed a utilisation of 60% on the web server and 30% on the application server at operating point, but the load on JBoss should be much lower because of the workmix error. New measurements showed a 60% load on the web server and 10% load on the application server.

How the problem was discovered and localised

This is what the MySQL log showed:

```
SELECT * FROM PAYMENT WHERE OWNER=null
SELECT * FROM ACCOUNT WHERE OWNER=null
```

OWNER should be an account number rather than null reference. Since we could manually log in and use the application, there should be no problem with the MySQL server, so there had to be a problem with either the application- or web-server, or the Grinder load generator.

The problem was localised with Grinder HTTP plugin that sniffed the interaction between Grinder and Tomcat. The sniffer showed that a user was logged in, but subsequent requests was performed as if the user was not logged in.

The solution

We had to change the first visited URL. Grinder will automatically get and use the first session-ID it finds for subsequent calls in one session. We have to perform a login request afterwards. If we do not, the BankApp application will start a new session for us with a different session-ID. This is what happened in the FL case.

Example on Grinder code that fixes the problem:

```
test1.GET("http://node05:8080/BankApp/index.jsp")
test2.POST('http://node05:8080/BankApp/transigo/login', self.data.login())
```

C.6 Tomcat threads

The built-in HTTP server in Tomcat is not optimal for production environments, but for development purposes it sufficient. To be able to utilise it to 70% we had to use 700 threads. This high number of threads was a problem on the old ClustIS

This was a problem with current setup, as Tomcat started one thread for each user session, resulting up to almost 700 concurrent java threads for the Tomcat server. The overhead was way too high on the old nodes.

We could solve the problem by simply shortening the session length for each user, but that would not reflect an actual user load. No one enters payment data manually in milliseconds. The KID number itself takes many seconds to write for a regular user.

In Tomcat configuration file, an attribute *maxKeepAlive* wase changed to "1". This means that we disable persistent connections, since each connection is used for only 1 request before it is closed. This leads to overhead in closing and opening HTTP connections, but removes the thread overhead on the web server, since the threads do not wait for new requests. After that change the number of java threads were in the range of 70 to 110, not 700 as it was in one case.

The new Clustis2 had a so-called O(1) kernel scheduler that minimized thread overhead. We could have changed the experiment setup to utilise persistent HTTP connections again, but it was not done.

C.7 IOstat disk utilisation

The first measurements performed were on Clustis1. IOstat did not report any utilisation for the disks, so we had to ignore this metric initially. On Clustis2, the second cluster, strange utilisations were reported from IOstat.

On the database server node, the utilisation were measured on a single disk for a given workload. Utilisations were measured to be ~0.2%, and did not increase when the workload was increased. This indicated that IOstat did not report the correct disk utilisation.

Fortunately, it turned out that we did not need to measure the disk utilisation. Web and application server does not use local disks, only shared storage when they load the software. After that everything runs in memory. Since we chose not to focus on the database we could ignore this problem.

C.8 Loading images with Grinder

To simulate the complete browser behaviour, the Grinder script should be configured to load images explicitly in addition to the actual HTML page. These images will be stored in the client cache and not downloaded again until they are expired. There was a problem with getting Grinder to check the expired-field of the html header, and the images were downloaded each time. The alternative was to ignore downloading the images completely.

The pictures for this application were approximately 1kB and 13kB in size, and we chose to ignore the images. After all, a separate static HTTP server should be responsible for handling images, not the dynamic web server.[1]

[1] On the other hand, if the images were a part of the user data, the images would become very important to deal with.

D Work on measuring integrated configuration

In Section 16.4 we proposed to add an RMI component to the SP model as further work. The reason is the massive overhead in network messaging between the web and application server when they reside on separate nodes. In this chapter that overhead is measured by single user stepwise measurements.

D.1 Integrated configuration

While it is convenient to separate the web and application server on two nodes for parameterising purposes, it may not be the best solution with performance issues in mind. It is therefore interesting to find the overhead caused by this separation.

The total overhead of separating the web and application server is identified as Java RMI overhead + network messaging overhead. RMI is "remote method invocation", where objects can be invoked from remote Java virtual machines.

The separate configuration is the baseline, and is measured in Chapter **??**. The integrated configuration is measured by reconfiguring the baseline system so that the web server runs in the same Java virtual machine as the application server. In practice, the integrated Tomcat servlet container is enabled on the JBoss application server, and does not need much special configuration.

D.2 Service demands

Service demand for the integrated web/application-server is measured with single-request-at-a-time tests described in Chapter 13. Each operation is measured with single request at-a-time. Consequently, the system is measured with only one user at any moment.

These results are then compared with the corresponding baseline parameter measurements. The "separate" column shows the sum of web and application server service demands, and the ratios between the two systems are computed.

The comparison shows clearly that the separate solution is not very effective with regard to performance. Where no service from the EJB layer is needed (on the application server), the response time is almost equal. When EJB services is needed the response time increases with a factor of 1.4 to 3.0. The variation is a result of different amounts of RMI calls and sizes.

Conside the workmix, where all requests are performed 3 times per init and logout. The total service demand for the workmix on the separate configuration is then:

$$Total\ service\ demand(separate) = 8,4 + 3(25,9 + 34,6 + 34,0 + 9,0 + 6,8 + 28,2) + 7,0 = 430,9$$

$$Total\ service\ demand(integrated) = 8,8 + 3(18,3 + 12,8 + 12,4 + 8,7 + 6,8 + 9,3) + 6,3 = 220,0$$

Operation	Separate	Integrated	Ratio (separate/integrated	Invokes EJB
Init	8,4	8,8	0,95	-
Login	25,9	18,3	1,42	X
Payment	34,6	12,8	2,70	X
PaymentDetails	34,0	12,4	2,74	X
NewPayment	9,0	8,7	1,03	-
ConfirmPayment	6,8	6,8	1,0	-
PaymentReceipt	28,2	9,3	3,03	X
Logout	7,0	6,3	1,11	-

Table D.1: Service demands for integrated web and application server

D.3 CPU usage

CPU utilisation is measured with multiple user tests.

The number of concurrent user sessions are increased in subsequent measurements until the CPU utilisation reaches 80%. After that, the response times increases drastically when adding more concurrent user sessions. At the operating point of 70% (note that it is , the capacity of the system is approximately a load of N=1870. Baseline measurements(separate configuration) gives a load of approximately N=1050 for the same operating point.

These numbers indicate that integrated solution with one node performs almost equal to a separate system with two nodes.

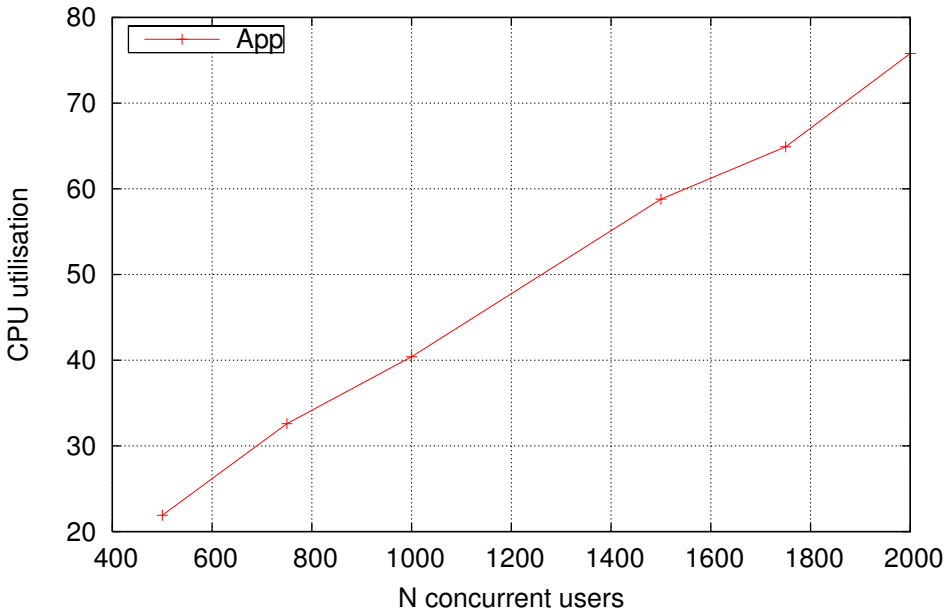


Figure D.1: CPU usage for integrated configuration

Comparing the figures for separate and integrated configurations we get these ratios:

Ratio, service demand measurement tests: $separate/integrated = 220,0/430,9 = 0,511$

Ratio, CPU utilisation measurement tests: $separate/integrated = 1030/1890 = 0,544$

Note that the service demand tests are single user measurements, while the CPU utilisation tests are multiple user measurements, introducing the possibility of increased overhead.

D.4 Session service demands

Service demands on integrated application server. The service demands are decreasing as the load increases. At load N=2000, the total CPU utilisation is 75%, and the garbage collector utilisation is 3%.

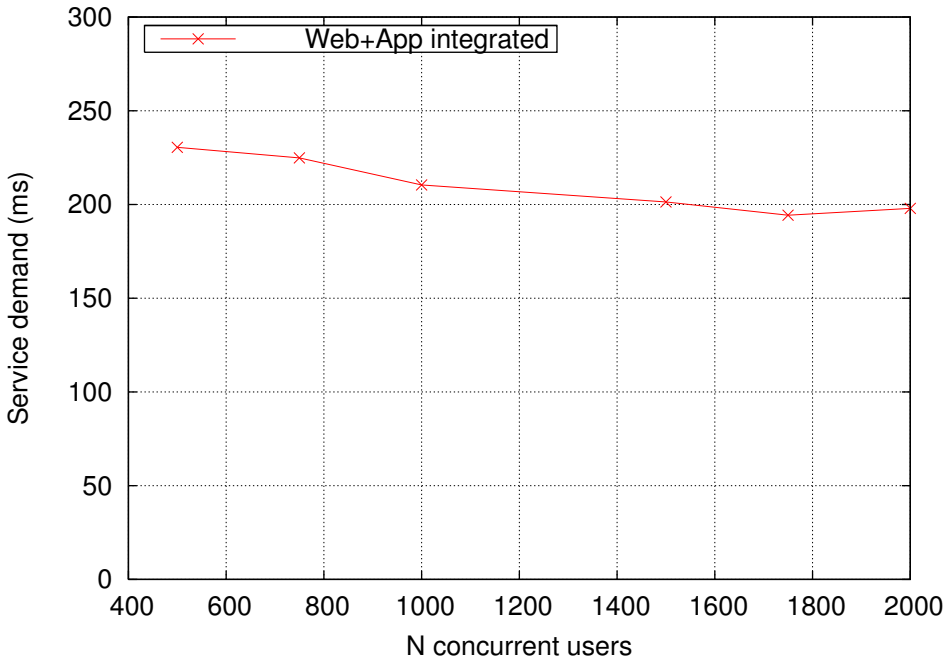


Figure D.2: Service demands on intergrated web and appserver

The integrated system does not reveal any measurable non-linear effects either.

E Work on accessing EJBs directly with Grinder

In the further work section, Section 16.4, we proposed to isolate the application server by omitting the web server. This can be done by accessing EJB's directly from Grinder. The purpose is to bypass the HTTP server and web server, in order to measure application server without having to set up many web servers to generate enough load on the application server.

E.0.1 run.sh

This batch file shows how to run the grinder-script ejb-access.py. The file also shows how to include the CLASSPATH, so the needed libraries can be imported into Jython.

```
1 #!/bin/sh
2
3 CLASSPATH="/home/geirbo/Measurements/jython/lib/jnp-client.jar:/home/geirbo/Measurements/jython/
              lib/jbossall-client.jar:/home/geirbo/Measurements/jython/lib/jb$:/home/geirbo/
              Measurements/jython/lib/BankApp.jar:/home/geirbo/Measurements/jython/lib/
              transigo_framework34.jar"
4
5 "/home/geirbo/scalability/sun-jdk-1.4.2.04/jre/bin/java" -Dpython.home="/home/geirbo/scalability/
              jython-2.1" -classpath "/home/geirbo/scalability/jython-2.1/jython.jar:$CLASSPATH" "
              org.python.util.jython" ejb-access.py   #"$@"
```

E.0.2 workmix.py

This file shows hwo to to access these EJB methods:

- customerEJB: *doLogin*
- accountEJB: *doGetAccounts*
- paymentEJB: *doGetPayments*, *doAddPayments*

```
1  # Script for testing workmix
2
3  from net.grinder.script import Test
4  from net.grinder.script.Grinder import grinder
5  from net.grinder.plugin.http import HTTPRequest
6  from HTTPClient import NVPair
7  from java.util import Random,Properties,Date
8  from java.lang import System,String
9  import os, os.path
10
11 from javax.naming import Context,InitialContext
12 from javax.rmi import PortableRemoteObject
13 from com.edb.transigo.bankapp.services.customer.business.vo import LoginCustomerVO
14 from com.edb.transigo.bankapp.services.payment.business.vo import PaymentVO
15 from com.edb.transigo.framework.control import ClientContext
16
17
18 dataFileName = "/tmp/geirbo/grinder/database-extract.out"
19 log = grinder.logger.output
20 randomStream = Random(System.currentTimeMillis())
21
22
```

```
### Environment for the JNDI context
env = Properties()
env[Context.INITIAL_CONTEXT_FACTORY] = "org.jnp.interfaces.NamingContextFactory"
env[Context.URL_PKG_PREFIXES] = "org.jboss.naming:org.jnp.interfaces"
env[Context.PROVIDER_URL] = "jnp://compute-0-1:1099"

### Reference to EJBs
context = InitialContext(env)

accountEJB    = context.lookup("java:ejb/AccountService").create()
customerEJB   = context.lookup("java:ejb/CustomerService").create()
paymentEJB    = context.lookup("java:ejb/PaymentService").create()
print customerEJB;print accountEJB;print paymentEJB


#Put business logic in functions
def login(userID, userPW):
  clientContext = ClientContext()
  clientContext.setUserId(userID)
  loginCustomerVO = LoginCustomerVO(userID, userPW)
  loginCustomerResponseVO = customerEJB.doLogin(clientContext, loginCustomerVO)
  #print loginCustomerResponseVO
  #print "Login:  UserID: " + userID + ",  customerID: " + loginCustomerResponseVO.firstName
  return clientContext

def getAccounts(clientContext, userID):
  accountVOList = accountEJB.doGetAccounts(clientContext, userID)
  #print accountVOList

def getPaymentDetails(clientContext, userID):
  paymentVOList = paymentEJB.doGetPayments(clientContext, userID)
  #print paymentVOList

def addPayment(clientContext, userID, payInfo):
  paymentVO = PaymentVO(payInfo[0], payInfo[1], payInfo[2], payInfo[3], payInfo[4], payInfo[5],
              payInfo[6], payInfo[7], payInfo[8], payInfo[9], payInfo[10], userID)
  paymentEJB.doAddPayment(clientContext, paymentVO)


#Wrap functions as Grinder tests
loginTest              = Test(1, "login customer").wrap(login)
getAccountsTest        = Test(2, "get accounts").wrap(getAccounts)
getPaymentDetailsTest  = Test(3, "get payment details").wrap(getPaymentDetails)
addPaymentTest         = Test(4, "add payment").wrap(addPayment)



class DataSource:
  def __init__(self):
    self.fileSize = os.path.getsize(dataFileName)
    self.dataFile = open(dataFileName,"r")

  def getRandomString(self):
          filePosition = randomStream.nextInt(self.fileSize) - 1000   # less than filesize to
                         avoid EOF

                self.dataFile.seek(filePosition)

                #Seek points to character position in file, not at beginning of newline
                #Read the rest of the line, so that next read line will be one full line
                line = self.dataFile.readline()
                line = self.dataFile.readline()

                #remove trailing newline (\n) character
                line = line.rstrip()

                #split line into tokes in an array
    data = line.split(';')

    return (data)

class TestData:
```

```
93    customer = ""
94
95    def __init__(self):
96      self.source = DataSource()
97      self.payment = 2
98      self.customer = self.source.getRandomString()
99
100   def loadNewCustomer(self):
101     self.customer = self.source.getRandomString()
102
103   def loginInfo(self):
104     customerID = self.customer[0]
105     password   = 'asdfasdf'
106     log("++++++++++ Customer: " + customerID)
107     return ([customerID, password])
108
109   def getPaymentInfo(self):
110     transactionDate = Date(System.currentTimeMillis())
111     amount = 1234.00
112     text = 'Payment text example'
113     fromAccount = self.customer[1] # [1] is placement of from account
114     toAccount = '98765432101'
115     customerNumber = '179280998861'
116     receiverName = 'receivers name'
117     receiverAddress = 'adress'
118     receiverZip = '1234'
119     receiverProvince = 'province'
120     KID = ''
121     id = '11111111111'
122     return ([transactionDate, amount, text, fromAccount, toAccount, customerNumber, receiverName,
                  receiverAddress, receiverZip, receiverProvince, KID, id])
123
124 class TestRunner:
125   def __init__(self):
126     self.data = TestData()
127
128   def __call__(self):
129
130     #Prepare customer data
131                 loginInfo = self.data.loginInfo()
132     userID = loginInfo[0]
133     userPW = loginInfo[1]
134     payInfo = self.data.getPaymentInfo()
135
136
137     #Workmix
138
139     grinder.sleep(10000)
140                 clientContext = loginTest(userID, userPW)
141
142                 for i in range(3):
143
144       grinder.sleep(15000)
145       getAccountsTest(clientContext, userID)
146
147                   grinder.sleep(10000)
148       getPaymentDetailsTest(clientContext, userID)
149
150                 for i in range(3):
151
152                         grinder.sleep(10000)
153
154       grinder.sleep(120000)
155
156       grinder.sleep(5000)
157       paymentInfo = self.data.getPaymentInfo()
158       addPaymentTest(clientContext, userID, paymentInfo)
159
160     grinder.sleep(15000)
161
162     ### Get new (random) customer from file, and wait a while.
163     self.data.loadNewCustomer()
```

```
164 |     grinder.sleep(10000)
```

E.1 Progress

The system was measured as a regular workmix measurement. The mean session time was still 515 seconds. Measurements were performend on upgrade nodes on Clustis2. A load of N=3400 user sessions was chosen, being able to compare with regular upgrade measurements. The heap was therefore set at 700MB.

JBoss showed a CPU utilisation of 6.3%, and the database CPU 2.6%. The garbage collector usage was 0.29% and reported a max heap usage of 80MB. The throughput was 19.8 tps.

The baseline multiple user measurements for N=3400 shows an application CPU usage of 25.0%, database 6.8%, and a max reported heap of 80MB.

The database utilisation for EJB measurements is only the half of the workmix measurements. The application server utilisation for EJB measurements is only the 1/5 of the workmix measurements, and the database utilisation only the half of the workmix measurements. Some effort was spent trying to locate what made the differences, but since the task was started late in the project, it was decided not to pursue it further.

The MySQL logs showed the same output for EJB measurements as for workmix measurements.

F Capturing network parameters

In this chapter we find the total network resource requirement pr. session to be 559kB. The result is obtained by using available information, and reading graphs that showed network input and output statistics for each node. Using the total response time from the web server to the user, combined with the network usage on the end node (database) we can calculate the bandwidth usage on the other nodes.

The network usage of 559kB per user session does not explain all unaccounted response time. Assuming a max utilisation of 0.5Gbit on the gigabit network interface, one session requires 9 milliseconds on the interface. Measurements show that there is 33 ms unaccounted for. This may be due to overhead in the network interface and that the max utilisation should lower.

Here we present the steps in obtaining the require network bandwidth per session. Each step in the list maps to the notes in square brackets in Figure F.1.

1. Using output from grinder logs to find the response lengths of the server responses. Calculating number of required IP packages (with overhead of 40 bytes pr. 1500).
2. Multiply the overhead with the workmix to obtain total response length pr. session
3. The session throughput is 1000 sessions for the baseline divided by the think time.
4. Read off the numbers from the graphs inSection 10.7 for LAN usage on each node.
5. Convert the graph readings to KB per (user) session by dividing with the throughput
6. Start with the end node (database)
7. For example: the graph reading for app server “out” means traffic to both web and db server. Since we know the database server “in” traffic, we can calculate the traffic to the web server.
8. Guessing the network usage from the user to the web server so that the other numbers fit.
9. Sum all “out” (to) traffic to get the total session bandwidth usage. (47+209+218+48+18+18)

			Protocol overhead: ~40 bytes pr. 1500bytes in package		
Workmix	Response lengt	IP packages	Overhead in	Workmix included	
1	9 758	6.7	10 025	10 025	[1]
1	9 860	6.8	10 130	10 130	
3	13 668	9.4	14 042	42 127	
3	10 839	7.4	11 136	33 408	
3	12 560	8.6	12 904	38 712	
3	11 198	7.7	11 505	34 514	
3	9 958	6.8	10 231	30 692	
1	9 381	6.4	9 638	9 638	
	Total response length pr. session			209 248 bytes	
				209 KB	[2]

Users:	1 000	Reported from Grinder
Think time:	515 s	Estimation/guessing
Throughput	1.94 tps [3]	Read from graph
		Calculated (KB pr user - from/to)

	[4]		[5]				
User:	graph readings		KB pr.user		Web		
in			209	from	209	[2]	
out			47	to	47		
Web	graph readings		KB pr.user		User	App	
in	185	KB/s	95	from	47	48	[7]
out	830	KB/s	427	to	209	218	
App	graph readings				Web	DB	
in	440	KB/s	227	from	218	8	[8]
out	130	KB/s	67	to	48	18	
DB	graph readings				App		
in	15	KB/s	8	from	8	[6]	
out	35	KB/s	18	to	18		
			1098				
				Total network usage	**559** KB		[9]

Network	0.50	Gbit	
bandwidth	500 000.00	Kbit	
	62 500.00	KB	
	63	KB/ms	
Load	560	KB	
Requires	**8.96**	ms	(minimum)

Figure F.1: Calculating network usage using a spreadsheet

G Work on transforming the SP model

Initially, this project was going to build on a complex SP model built by Fagerlie-Landmark [9]. Assuming that there were non-linear effects, a lot of time were spent extending the modelling work of Fagerlie-Landmark by restructuring and transforming the SP model. But measurement results showed that there were no detectable non-linear effects. Note that this result is only valid for the range of measurements performed, and is also limited to the work-definition we use. There was no reason to keep the complex SP model.

In this chapter we show some of the work from the process of improving or transforming the model. This was also a process of exploring the possibilities and challenges of SP modelling, namely using it in an object-oriented design world. The motive is that little work has been done yet to incorporate object oriented design in SP.

G.1 Simple model

The SP model presented in this thesis is very simple. Each high level software component corresponds to one SP component, and each of these components maps to a separate physical computer node.

An SP model should be as simple as possible, an this is almost the simplest model for this application context.

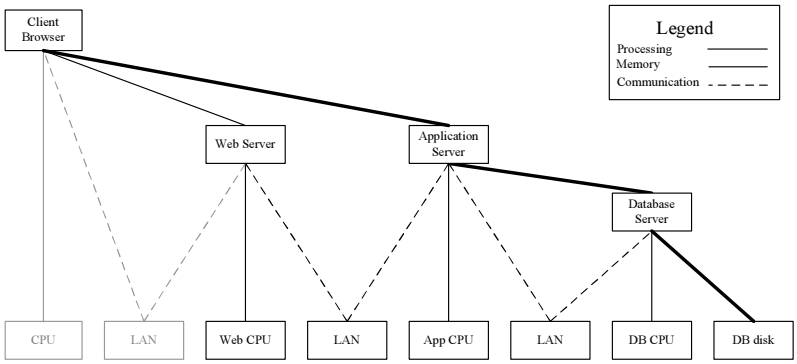


Figure G.1: SP - simple model

G.2 Fagerlie-Landmark model

The model presented is constructed by Fagerlie-Landmark and used in their diploma work[9]. It is a result of many iterations of grouping and de-grouping classes of software. The scope in their diploma was not as general as in this thesis, so the model was tailored for that specific system. It is built pretty much ad hoc-style and the process of getting there is not documented.

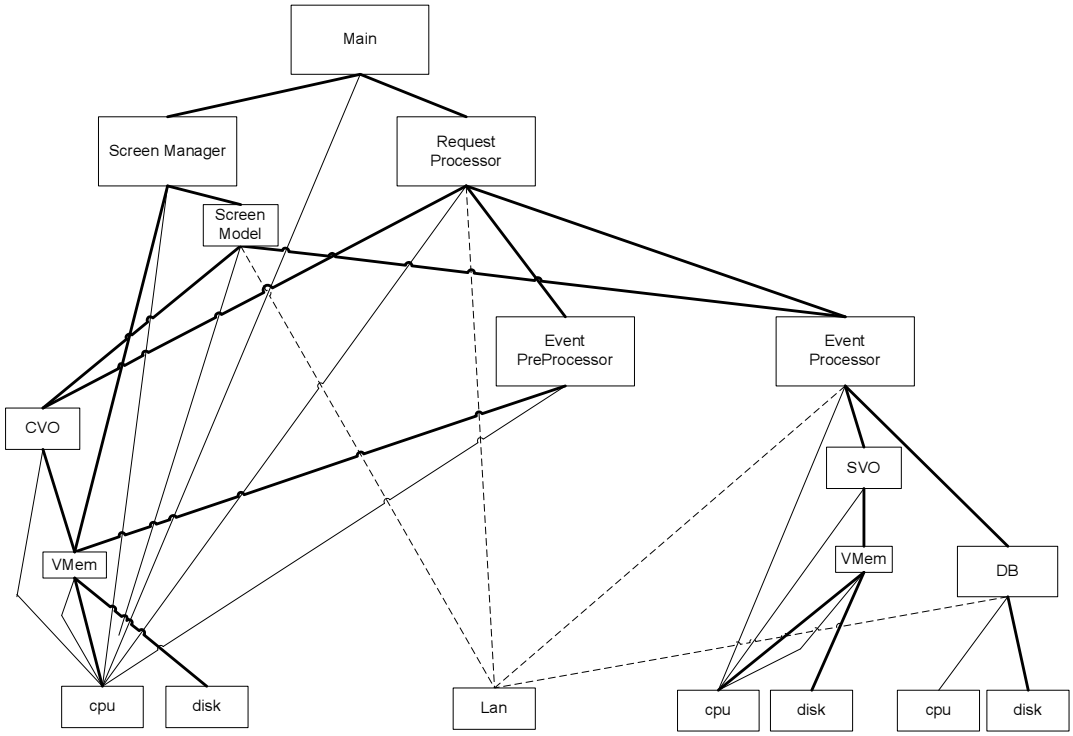


Figure G.2: Fagerlie-Landmark SP model

G.3 Intuitivity

Two types of intuitivity are identified when working with transforming SP models, namely "conceptual intuitivity" and "developer intuitivity".

Developer intuitivity The FL model is built primarily to map the structure of the system in terms av Java methods and classes. In that sense it is intuitive, at least for a developer that sees the system through the code. But conceptually the model is harder to comprehend. Even with lots of experience working with and reading all kinds of high-level models, this model is hard to understand since you have to follow the method calls and objects being sent back and forth.

Conceptual intuitivity Conceptual intuitivity is when a model is built to be understandable on a conceptual plan. It is easy for a person to understand such a model, even though he do not know the system.

The new models described in the next sections are intuitively more understandable on a conceptual plan, but is not that intuitive with regards to java method invocation. A developer may have problems relating the code to the model.

G.4 MVC in SP

When modelling the Model View Controller pattern (MVC) in SP we have to define a hierarchy. Messages is passed back and forth from *model* and *view*, see *Figure G.3*. View queries model, but Model

notifies View when changes in underlying data changes, thus causing View to do work. SP does not allow work to be put back and forth between components, work is always inflicted downwards the tree graph.

We define method invocations(state query) as work being put on a component down the hierarchy. Events (change notification) are not shown directly in the model, but are instead represented in the complexity matrices from *controller* to *model* or *view* as the probability that an notification has occurred, based on what transaction type is executed.

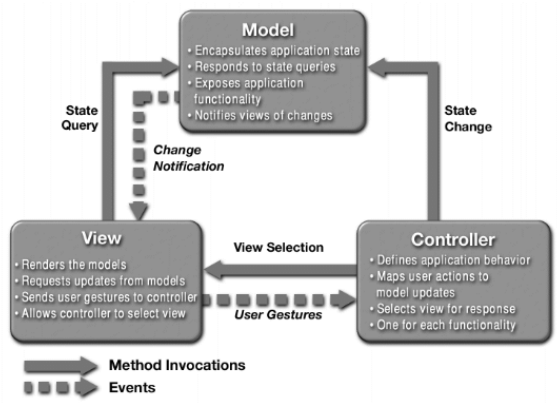


Figure G.3: Model View Controller (SUN blueprints)

G.4.1 MVC model

Two important J2EE design patterns were emphasized in the figure. The reason was to make a general model that can be reused more easily.

The Model View Controller pattern(MVC) was a natural choice to start with. In this model Main component is the *controller*, and ScreenManager has the *view* role. ModelManager is *model*. The *view* in MVC states requests to the *model* when data is needed for presentation, and the *controller* states changes to the system.

The second design pattern incorporated in the model is transfer objects. The old/deprecated name for them were *value objects.*

We have simplified the model somewhat, hiding hardware devices, operating system and Java virtual machine in a black box. The J2EE layer and classes are also hidden in that black box.

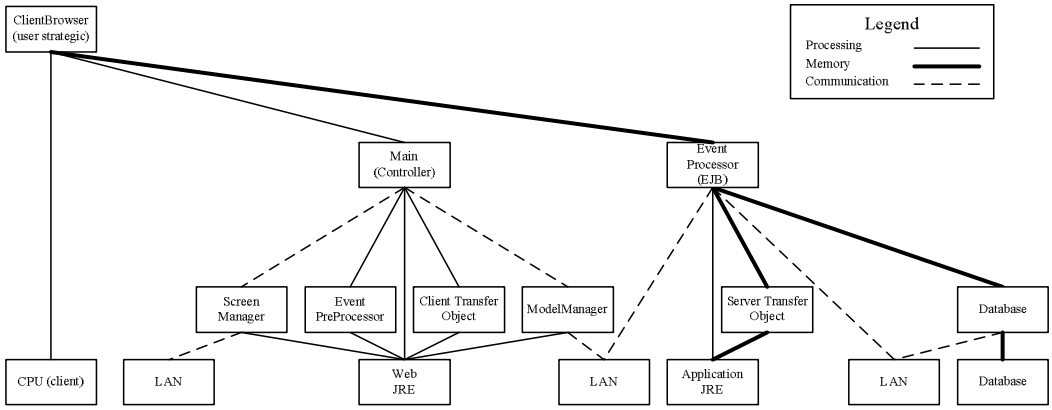


Figure G.4: SP - collapsed view of a transformed model

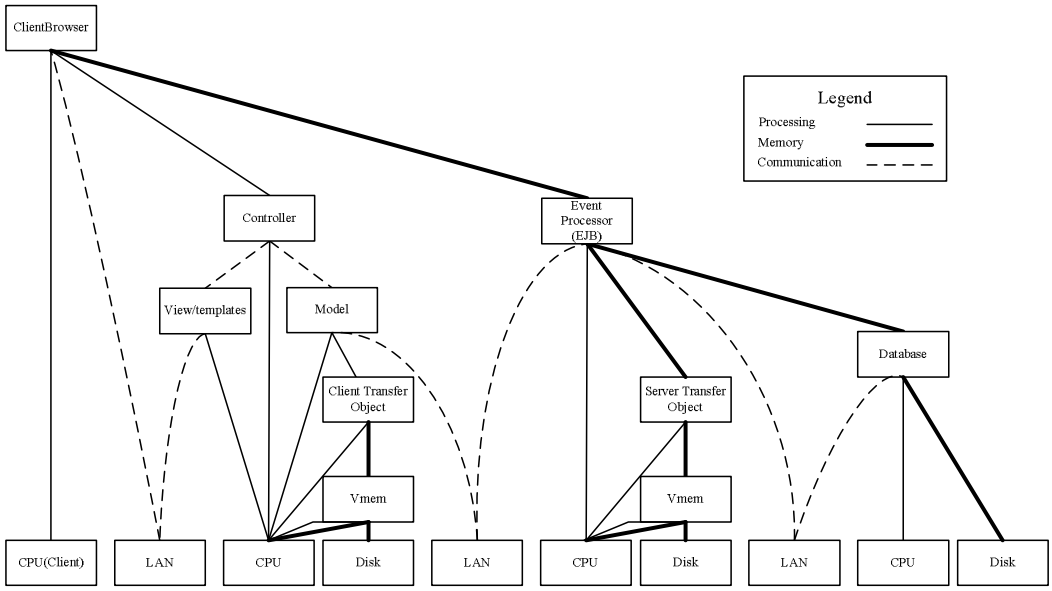


Figure G.5: SP - complete view of a transformed model

This is the complete view of the model, where the Java Run Time Environment (JRE) and operating system components have been decomposed.

The new model is less developer-intuitive, meaning that some Java method invocations between classes are not represented as links between components. The most obvious example is the link between the web and the application tier.

H Toolbox load generation

In this chapter we implement the workload from Chapter 8. The load scripts are a part of the toolbox.

H.1 Grinder workload script

This Python script is originally written by Fagerlie&Landmark in [9]. Each Grinder thread will run one instance of this script. The script simulates a user that logs in, perform requests and log out. When the script is finished, the Grinder thread will run the script again. The last line of the script prepares a new and random user to be loaded.

The script has two important modifications from the original version:

- Login info was stored in the script itself, but is now stored in a file. The file has 1 line for each user that can be logged in to the system. A new class DataSource was created. Login info is account numbers and payment IDs, which are used when emulating a user logging in to the system. The login info file is automatically generated when populating the database with users.
- Originally the script did not actually log in any users, since a crucial HTTP request was missing. The request is necesarry to obtain a sessionID from the Tomcat server, and that sessionID is used in all subsequent calls for that user session:

Listing H.1: Capture sessionID

```
110     test1.GET("http://compute-1-0:8080/BankApp/index.jsp")
```

H.1.1 workmix.py

Listing H.2: Grinder load script

```
1  # Script for testing workmix
2  from net.grinder.script import Test
3  from net.grinder.script.Grinder import grinder
4  from net.grinder.plugin.http import HTTPRequest
5  from HTTPClient import NVPair
6  from java.util import Random
7  from java.lang import System
8  import os, os.path
9
10 dataFileName = "/lwork/geirbo/grinder/database-extract.out"
11 log = grinder.logger.output
12 randomStream = Random(System.currentTimeMillis())
13 test1 = Test(1, "request main page").wrap(HTTPRequest())
14 test2 = Test(2, "login").wrap(HTTPRequest())
15 test3 = Test(3, "payments").wrap(HTTPRequest())
16 test4 = Test(4, "payment details").wrap(HTTPRequest())
17 test5 = Test(5, "new payment").wrap(HTTPRequest())
18 test6 = Test(6, "confirm payment").wrap(HTTPRequest())
19 test7 = Test(7, "payment receipt").wrap(HTTPRequest())
20 test8 = Test(8, "logout").wrap(HTTPRequest())
21 test9 = Test(9, "newCust").wrap(HTTPRequest())
22 test10 = Test(10, "newCustProc").wrap(HTTPRequest())
```

```
test11 = Test(11, "newAccount").wrap(HTTPRequest())
test12 = Test(12, "newAccountProc").wrap(HTTPRequest())

class DataSource:
  def __init__(self):
    self.fileSize = os.path.getsize(dataFileName)
    self.dataFile = open(dataFileName,"r")

  def getRandomString(self):
    filePosition = randomStream.nextInt(self.fileSize) - 1000  # less than filesize
    self.dataFile.seek(filePosition)
    #Seek points to character position in file, not at beginning of newline
    #Read the rest of the line, so that next read line will be one full line
    line = self.dataFile.readline()
    line = self.dataFile.readline()

    #remove trailing newline (\n) character
    line = line.rstrip()
    #split line into tokes in an array
    data = line.split(';')
    return (data)

class TestData:
    customer = ""

    def __init__(self):
      self.source = DataSource()
      self.payment = 2
      self.customer = self.source.getRandomString()

    def loadNewCustomer(self):
      self.customer = self.source.getRandomString()

    def login(self):
      log("++++++++++ Customer: " + self.customer[0])
      return (NVPair('customerID',self.customer[0]),#customer id placement
                              NVPair('password', 'asdfasdf'),)
    def getPayment(self):
      if self.payment >= 12:#12 is index of last payment in customers list
              self.payment=2
      self.ret = self.payment
      self.payment+=1
      return (NVPair('id',self.customer[self.ret]),NVPair('x','15'),NVPair('y','7'),)

    def getPaymentConfirm(self):
      return (NVPair('text','Payment text example'),
        NVPair('transactionDate','26.05.2003'),
        NVPair('receiverName','receivers name'),
        NVPair('receiverAddress','adress'),
        NVPair('receiverZip','1234'),
        NVPair('receiverProvince','province'),
        NVPair('fromAccount',self.customer[1]),#placement of from account
        NVPair('KID',''),
        NVPair('kroner','1234'),
        NVPair('ore','00'),
        NVPair('toAccount','98765432101'),
        NVPair('action.x','40'),NVPair('action.y','7'),)

    def newCustomer(self):
      tallsekvens =''
      #r = Random(System.currentTimeMillis())
      for i in range(11):
        tall = randomStream.nextInt(10)
        tallsekvens = tallsekvens + str(tall)
      return (NVPair('id',tallsekvens),
        NVPair('firstName','fornavn'),
        NVPair('lastName','etternavn'),
        NVPair('street','asdf'),
        NVPair('zip','1234'),
        NVPair('city','asdf'),
        NVPair('password1','asdfasdf'),
        NVPair('password2','asdfasdf'),)
```

```
95
96      def newAccount(self):
97         tallsekvens ='5221'
98         #r = Random(System.currentTimeMillis())
99         for i in range(7):
100          tall = randomStream.nextInt(10)
101          tallsekvens = tallsekvens + str(tall)
102        return (NVPair('newAccount',tallsekvens),NVPair('accountName','konto'),)
103
104 class TestRunner:
105   def __init__(self):
106    self.data = TestData()
107
108   def __call__(self):
109     #test1.GET("http://compute-1-0:8080/BankApp")
110     test1.GET("http://compute-1-0:8080/BankApp/index.jsp")
111     grinder.sleep(10000)
112     test2.POST('http://compute-1-0:8080/BankApp/transigo/login', self.data.login())
113     for i in range(3):
114       grinder.sleep(15000)
115       test3.GET('http://compute-1-0:8080/BankApp/transigo/payments')
116       grinder.sleep(10000)
117       test4.POST('http://compute-1-0:8080/BankApp/transigo/paymentdetails',self.data.getPayment()
                    )
118     for i in range(3):
119       grinder.sleep(10000)
120       test5.GET('http://compute-1-0:8080/BankApp/transigo/newPayment')
121       grinder.sleep(120000)
122       test6.POST('http://compute-1-0:8080/BankApp/transigo/confirmPayment',self.data.
                    getPaymentConfirm())
123       grinder.sleep(5000)
124       test7.POST('http://compute-1-0:8080/BankApp/transigo/paymentReceipt',(NVPair('add.x','40'),
                    NVPair('add.y','9'),))
125     grinder.sleep(15000)
126     test8.GET('http://compute-1-0:8080/BankApp/transigo/main')
127     ### Get new (random) customer from file, and wait a while.
128     self.data.loadNewCustomer()
129     grinder.sleep(10000)
```

H.2 Database generation scripts

This script creates a database filled with random data. The script automates all steps of the generation. The script must be run one time for each database size. The size of the database is set with the variable *TOTAL_CUSTOMERS*.

First the database is started. The database is reset and indexed, and then runs a Java program to generate and fill the database with randomised users.

The database files are copied from the node that runs the database, and archived into one file. The database arhive file is then put in the correct folder in the toolbox framework.

Finally we extract some info from the database and put it in a text file. We extract the login ID and account IDs, and this information is used by the Grinder scripts so that they can emulate the users.

Listing H.3: generate-database.sh

```
1  #!/bin/bash
2  CURRENT_RUN_DIR=`pwd`
3  cd `cat $HOME/.current_run_path`; source configuration.sh   #load config
4  cd $CURRENT_RUN_DIR
5
6  MYSQL_HOSTNAME="comp-pvfs-0-1"
7  TOTAL_CUSTOMERS=100000
8  THREADS_CONNECT=10
9  USER=$DB_USERNAME
10 PASSWORD=$DB_PASSWORD
11 DB_FOLDER="$LOCAL_NODE_DIR/mysql"
12 EXTRACT_FILE="$GRINDER_DATA_FILE"
```

```
13
14 let CUSTOMERS=$TOTAL_CUSTOMERS/1000
15 SAVE_FOLDER="$SERVERS_DIR/mysql/users_"$CUSTOMERS"K"
16 echo $SAVE_FOLDER
17
18
19 ### Initialise files and folders
20 mkdir $SAVE_FOLDER
21
22 ### Start database
23 ssh $MYSQL_HOSTNAME  "cd $SCRIPTS_DIR;  ./start-mysql.sh"  &
24 sleep 10
25
26 ### Reset database
27 echo ">>> Resetting database <<<"
28 mysql -h $MYSQL_HOSTNAME -u $USER -p$PASSWORD bankapp < bankapp_reset.sql
29
30 ### Index database
31 mysql -h $MYSQL_HOSTNAME -u $USER -p$PASSWORD bankapp < bankapp_indexing.sql
32
33 ### Generate database: Run java program
34 echo ">>> Generating database <<<"
35
36 cd $CURRENT_RUN_DIR/generate_database
37
38 export CLASSPATH=mysql-connector-java-3.1.7-bin.jar:.
39 javac FillDB.java
40 time java FillDB $MYSQL_HOSTNAME $TOTAL_CUSTOMERS $THREADS_CONNECT
41
42
43 ### Archive database files for measurement use
44 echo ">>> Archive database files <<<"
45
46 ssh $MYSQL_HOSTNAME "cd $DB_FOLDER; rm *.log; sleep 2; tar zcvf $SAVE_FOLDER/mysql.tar.gz *"
47
48
49 ### Generate database-extract: Run java program
50 echo ">>> Generating database extract file<<<"
51
52 cd $CURRENT_RUN_DIR/generate_database_extract
53
54 export CLASSPATH=mysql-connector-java-3.1.7-bin.jar:.
55 javac Data.java
56 time java -Xms800m -Xmx800m Data $MYSQL_HOSTNAME $EXTRACT_FILE
57 tar zcvf $SAVE_FOLDER/$EXTRACT_FILE.tar.gz $EXTRACT_FILE
58
59
60 ### Show results
61 #mysqlshow  -v -h $MYSQL_HOSTNAME -u $USER --password=$PASSWORD  bankapp ACCOUNT
62 #mysqlshow  -v -h $MYSQL_HOSTNAME -u $USER --password=$PASSWORD  bankapp PAYMENT
63 mysqlshow  -v -h $MYSQL_HOSTNAME -u $USER --password=$PASSWORD  bankapp CUSTOMER
64
65
66 ### Stop database server
67 ssh $MYSQL_HOSTNAME  "cd $SCRIPTS_DIR;  ./kill-processes.sh"     &
68
69
70
71 ### View save-folder
72 ls -l $SAVE_FOLDER/
```

I Toolbox scripts

This chapter contains some important toolbox scripts.

The toolbox is described in Chapter 9.

I.1 analyse-sar-log.pl

This program extracts utilisation data from the Sysstat SAR log files. The script analyses a file given as argument. The arguments *start* and *duration* specifies when in the experiment to analyse data, and for how long.

With regular expressions we obtain the time for each log entry. If the time is in the interval we want to analyse, we extract the CPU utilisation. We accumulate the utilisation, and counts the number of samples. With this we can calculate the mean utilisation for that interval.

Listing I.1: analyse-sar-log.pl

```
#!/usr/bin/perl
use Time::Local;

# Purpose:
#       Calculate the average CPU utilisation from sysstat log files.
#       Use the first time stamp in the log as time 0
# Command line parameters:
#       <file>  <start time (sec)>  <duration (sec)>
# Author: Geir Bostad, NTNU  (04.07.2006)
#

### Get parameters from command line
if ( $#ARGV < 2 )  { die "Error! Needs 3 Parameters to run script:\n <file>  <start time>  <
            duration time>\n" }

$FILE      = @ARGV[ 0 ];
$start     = @ARGV[ 1 ];
$duration  = @ARGV[ 2 ];

$month = 1; $day = 1; $year = 1;     # or else we get error "Day '' out of range"
$sum = 0; $avg = 0; $count = 0; $value = 0; $time_shift = 0;

open(FILE) or die("Could not open file: $FILE");
foreach $line (<FILE>) {

  # Get date. Log file format:
  # "Linux 2.6.9-5.0.5.ELsmp (clustis2.idi.ntnu.no)  07/04/2006":
  if ( $line =~ /\w+.* (\d+)\/(\d+)\/(\d+)$/ ) {
    $month = $1-1;   # Perl months are [0..11]
    $day   = $2;
    $year  = $3;
    #print "$month $day $year  ";
  }

  # Match lines on format:  (Last coloumn shows CPU idle time)
  # "09:24:24 PM  all  18.00  0.00  2.40  0.00  79.60"
  if ( $line =~ /^(\d+):(\d+):(\d+)\s\w+\s+all(\s+\d+\.\d+){4}\s+(\d+\.\d+)/ ) {

    $log_time = timelocal($3, $2, $1, $day, $month, $year) + $time_shift;
    $value = 100 - $5 ;  # Converting from CPU idle time to utilisation
```

```
40
41      # Check if a new 'day' occured: "12:59:56" AM to "01:00:01 AM"
42      if ( $log_time <$last_time ) {
43        $time_shift += 12*3600;
44        $log_time = $log_time + $time_shift;   #changing the current sample also, or else we loose
                    it
45      }
46      $last_time = $log_time;
47
48      # Use first time stamp in the log as time 0.
49      # After the '$start_time' variable is initialized, this code block won't run again.
50      unless ( $start_time ) {
51        $start_time = $log_time + $start;
52        $stop_time  = $log_time + $start + $duration;
53      }
54
55      if ( ($start_time <= $log_time) && ( $log_time <= $stop_time) ) {
56        # Accumulate utilisation samples from log file
57        $sum = $sum + $value;
58        $count++;
59      }
60    }
61  }
62
63  $average = 0;
64  if ( $count > 0 ) {
65    $average = $sum / $count;
66  }
67
68  print sprintf "  Utilisation:%6.2f%  (%4d samples, sum:%6.1f)\n",  $average, $count, $sum;
```

I.2 analyse-grinder-logs.pl

This script processes Grinder log files to extract average transaction response times for each request. Data are extracted for a given time interval, specified as seconds in *START_TIME* and *LENGTH* arguments.

Grinder reports statistics with one file per node. This script parses these files. From the log files we obtain the time when a request was issued, what type of request it was and its response time. The results are accumulated in an associative array, one file at a time.

The average of response time is calculated for each type of request. Remember that each type corresponds to an element in the workmix matrix.

The workmix vector is automatically derived from the logs. If the workmix does not compare to our defined workmix of [1,1,3,3,3,3,3,1] then there may be a problem with the measurement.

The script uses the workmix is used to calculate the mean total user session response time. The throughput is calculated by dividing the number of customers that left the system (request type 8), by the length of the interval that we analysed data in.

Listing I.2: analyse-grinder-logs.pl

```
1  #!/usr/bin/perl
2  use Time::Local;
3
4
5  # Function of script:
6  #     Calculate the average and total response time for each request from grinder log files.
7  #
8  # Command line parameters:
9  #     <path-to-logfiles>   <start time (sec)>  <interval time (sec)>
```

```
#
# The format on Grinder log files:
#     Grinder log files format: (filename example: data_compute-0-18.local-0.log)
#     Thread, Run, Test, Milliseconds since start, Transaction time, Errors, HTTP Response Code,
          HTTP Response Length, HTTP Response Errors
#     Example: "342, 0, 1, 422, 57, 0, 200, 9758, 0"


# Get parameters from command line
if ( $#ARGV < 2 )  { die "Error! Needs 3 Parameters to run script:\n <path-to-logfiles>  <start
          time>  <duration time>\n" }
$path               = @ARGV[ 0 ];
$start_time_str     = @ARGV[ 1 ];
$duration_time_str  = @ARGV[ 2 ];


# Converting seconds to milliseconds
$start_time = $start_time_str * 1000;
$duration_time = $duration_time_str * 1000;
$stop_time = $start_time + $duration_time;



### Get list of files to be analysed
opendir(DirHandle, $path) or die("Can't access directory: $path");
@file_list = grep /^data.*/, readdir(DirHandle);
closedir(DirHandle);


# results: Hash of hashes:
# testid1 : (sum1, samplecount1, average1, ratio1, workmix1)
# testid2 : (sum2, samplecount2, average2, ratio2, workmix2)



### Parse log files
foreach $file (@file_list) {

  # Process one log file at a time
  chdir "$path" or die ("Can't open dir $path");
  open(FILE, "$file") or die("Could not open file: $file");
  foreach $line (<FILE>) {

    if ( $line =~ /^\d+, \d+, (\d+), (\d+), (\d+)/ ) {
      $id = $1;
      $time = $2;
      $value = $3;

      if ( ($time >= $start_time) && ( $time < $stop_time) ) {
        #print "Time: $time   Type: $id   response: $value\n";
        $results{$id}{sum} += $value;
        $results{$id}{sample_count}++;
        push @{ $results{$id}{sample_list} }, $value;
      }
    }
  }
  close(FILE);
}


### Calculate standard deviation
sub mean {
  my $result;
  foreach (@_) { $result += $_ }
  return $result / @_;
}
sub std_dev {
  my $mean = mean(@_);
  my @elem_squared;
  foreach (@_) {
    push (@elem_squared, ($_ **2));
  }
```

```
80    return sqrt( mean(@elem_squared) - ($mean ** 2));
81  }
82
83  for $id ( keys %results ) {
84    #foreach $el (@ { $results{$id}{sample_list} }) {  print "$el " }
85    $results{$id}{std_dev} = std_dev( @ { $results{$id}{sample_list} } );
86  }
87
88
89
90
91  ### Derive workmix vector from logs.
92  ### Find the smallest instance of 'samples'. Then find the ratio between the tests.
93
94  $min_samples_id = 1;            # first assume that test 1 is smallest
95  for $id ( keys %results ) {
96    if ( $results{$id}{sample_count} < $results{$min_samples_id}{sample_count} ) {
97      $min_samples_id=$id;
98    }
99  }
100 for $id ( keys %results ) {
101   $results{$id}{ratio} =  $results{$id}{sample_count} / $results{$min_samples_id}{sample_count};
102   $results{$id}{workmix} = sprintf "%.0f", $results{$id}{ratio};    # Round to nearest integer
103 }
104
105
106
107 ### Calculate average of response times
108 for $id ( keys %results ) {
109
110   $results{$id}{average} = 0;
111   # Avoid zero division
112
113   if ( $results{$id}{sample_count} > 0 ) {
114     $results{$id}{average} = $results{$id}{sum} / $results{$id}{sample_count};
115   }
116 }
117
118
119 ### Calculate total response time
120 $total_response = 0;
121 for $id ( keys %results ) {
122   $total_response += ( $results{$id}{average} * $results{$id}{workmix} );
123 }
124
125
126
127 ### Print data for each test
128 print "Test  Mean(ms)      #samples    Ratio  Workmix     Sum   Std.dev\n";
129 for $id ( sort keys %results ) {
130   $test    = sprintf "%3u",   $id;
131   $count   = sprintf "%6u",   $results{$id}{sample_count};
132   $average = sprintf "%7.1f",   $results{$id}{average};
133   $sum     = sprintf "%8u",   $results{$id}{sum};
134   $ratio   = sprintf "%5.3f", $results{$id}{ratio};
135   $workmix =                  $results{$id}{workmix};
136   $stddev  = sprintf "%6.2f", $results{$id}{std_dev};
137
138
139   #print "Test#$test: $average   ($count samples,  ratio:$ratio,  workmix:$workmix  sum:$sum,
                std.dev:$stddev)\n";
140   print "$test    $average       $count     $ratio     $workmix   $sum    $stddev\n";
141 }
142
143 ### Print total response time
144 $total_str = sprintf "%5u", $total_response;
145 print "\n";
146 print "Total response time:  ${total_str} ms  (workmix multiplied with mean response times)\n";
147
148
149 ### Print throughput
150 $departures = $count;   # $count refers to the last test we printed: logout(test8)
```

```
151 $throughput = $departures / $duration_time * 1000;
152 $throughput_str = sprintf "%5.3f", $throughput;
153 print "Throughput:           ${throughput_str} tps (users leaving system)\n";
```

Here we show the output when running the script:

Listing I.3: Output from analyse-grinder-logs.sh

```
Mean response times:
Test   Mean(ms)       #samples     Ratio   Workmix      Sum    Std.dev
  1         7.7           9688     1.000         1    74765      33.18
  2        74.3           9687     1.000         1   719387     100.92
  3        46.7          29092     3.003         3  1359435      77.17
  4        46.6          29094     3.003         3  1354393      74.33
  5         8.4          29077     3.002         3   244275      34.31
  6         9.1          29073     3.001         3   264529      32.91
  7        38.4          29068     3.001         3  1116277      69.50
  8         6.9           9691     1.000         1    67207      29.81

Total response time:    536 ms  (workmix multiplied with mean response times)
Throughput:           4.846 tps (users leaving system)
```

I.3 analyse-gc-log.pl

This script analyses the garbage collector log file. It takes a log file as argument, in addition to the start time and duration of the interval to be analysed.

The script iterates through all lines in the file, but calculates only results for the parts that are inside the interval we want to analyse.

The garbage collection times are accumulated into a total service demand and divided by the sample interval time to obtain the CPU utilisation. The script also reports number of full and tenured garbage collections, see [24] for a description of various types of garbage collections. The script calculates the average heap usage, and records the max heap usage over the sample interval.

Listing I.4: analyse-gc-log.pl

```
1  #!/usr/bin/perl
2  use Time::Local;
3
4  # Function of script:  Get the average garbage collection CPU usage from gc log files.
5  # Command line parameters:  <file>  <start time(sec)>  <duration time(sec)>
6
7
8  sub mean {
9    my $result;
10   foreach (@_) { $result += $_  }
11   return $result / @_;
12 }
13
14 sub max {
15   my $temp_max = 0;
16   foreach (@_) {
17     $value = $_;
18     if ($value > $temp_max ) { $temp_max = $value; }
19   }
20   return $temp_max;
21 }
22
23 ### Get parameters from command line
24 if ( $#ARGV < 2 )  { die "Error! Needs 3 Parameters to run script:\n <file>  <start time(sec)>  <
       duration time(sec)> \n" }
25
26 $FILE          = @ARGV[ 0 ];
27 $start_time    = @ARGV[ 1 ];
```

```
28 $duration_time = @ARGV[ 2 ];
29
30 if ( $duration_time < 1 ) { die "Error! Duration time must be at least 1 second \n" }
31
32 $sample_time = 0;
33 $cur_time    = 0;
34 $stop_time   = $start_time + $duration_time;
35
36
37 $sum          = 0;
38 $samplevalue  = 0;
39 $samplecount  = 0;
40 $majorcount   = 0;
41 $tenuredcount = 0;
42 $heapsize     = 0;
43 @heapsize_list = ();
44
45 #Format in gc log files: (two minor: 'GC', one major: 'Full')
46 #1527.082: [GC 1527.085: [DefNew: 68713K->436K(76800K), 0.0028030 secs] 83332K->15055K(759488K),
       0.0028520 secs]
47 #2621.896: [GC 2621.896: [DefNew: 9820K->9820K(10240K), 0.0000110 secs]2621.896: [Tenured: 81283K
       ->70345K(91072K), 0.4354690 secs] 91104K->70345K(101312K), 0.4355330 secs]
48 #2371.254: [Full GC 71.254: [Tenured: 8421K->7904K(91072K), 0.0853240 secs] 17602K->7904K(101312K
       ), [Perm : 10901K->10901K(16384K)], 0.0853820 secs]
49
50 open(FILE) or die("Could not open file: $file");
51 foreach $line (<FILE>) {
52
53   if ( $line =~ /^(\d+\.\d+): .* (\d+\.\d+) secs]$/ ) {
54     $cur_time    = $1;
55     $samplevalue = $2;
56
57     # Get heap size before collection  (pattern matches minor collections)
58     if ( $line =~ /(\d+)K->\d+K\(\d+K\), \d+\.\d+ secs]$/ ) {
59       $heapsize = $1;
60     }
61
62     if ( ($cur_time >= $start_time) && ($cur_time <= $stop_time) ) {
63       #print "val: $samplevalue\n";
64       $sum = $sum + $samplevalue;
65       $samplecount++;
66       $last_time = $cur_time;
67       push @heapsize_list, $heapsize;
68
69       #Count the types of gc activity (major(full))
70       if ( $line =~ /Full GC/ ) {
71         $majorcount++;
72       }
73       elsif ( $line =~ /Tenured/ ) {
74         $tenuredcount++;
75       }
76     }
77   }
78  }
79
80 if ( scalar(@heapsize_list)< 1 ) { die "Error no elements matched time interval" }
81
82 $sample_time  = $last_time - $start_time + 1;     #In case measured time is shorter than specified
       duration
83 $average      = $sum / $sample_time * 100;        #Convert to percent (*100)
84 $average_heap = mean( @heapsize_list ) / 1000;    #Convert to MB from K  Note: This is only sample
       mean, not weighted avg
85 $max_heap     = max ( @heapsize_list ) /1000;
86
87 ### Print
88 $average_str       = sprintf "%5.2f", $average;
89 $count_str         = sprintf "%4d",   $samplecount;
90 $majorcount_str    = sprintf "%2d",   $majorcount;
91 $tenuredcount_str  = sprintf "%1d",   $tenuredcount;
92 $sample_time_str   = sprintf "%3d",   $sample_time;
93 $sum_str           = sprintf "%5.2f", $sum;
94 $heap_avg_str      = sprintf "%3u",   $average_heap;
```

```
95 $heap_max_str      = sprintf "%3u",   $max_heap;
96
97 print "  GC CPU usage:$average_str%  ($count_str samples [full:$majorcount_str, tenured:
       $tenuredcount_str] sum:$sum_str sec, sample time:$sample_time_str sec,  heap[avg:${
       heap_avg_str}MB,max:${heap_max_str}MB]\n";
```

This is an example of the output from the script:

Listing I.5: Output from analyse-gc-log.sh

```
Tomcat:
  GC CPU usage: 1.51%  (1071 samples [full:33, tenured:0] sum:30.15 sec, sample time:1999 sec,
                heap[avg:146MB,max:158MB]
JBoss:
  GC CPU usage: 0.30%  ( 404 samples [full:34, tenured:0] sum: 6.05 sec, sample time:1999 sec,
                heap[avg: 78MB,max: 79MB]
```

J Simulating the dynamic model

In Section 14.5 we presented the results from simulating the dynamic model. Here we show how the simulation is implemented.

We have implemented the BankApp model as a an open multiclass simulation. We calculate the arrival rate for each request, so all requests are started independently of each other. Hence it is not a step-wise load, and the simulation has a very short warm-up phase. The simulation produces the CPU utilisation of the servers.

This is a list of parts that are not implemented yet. Simulation is not our focus, we have only shown how to implement a simulation

- Network parameters. It can be added as an extra element in the parameters-array as "parameters[3]"
- Application garbage collection complexity function.
- Statistical object to obtain session response time

J.1 Open multiclass simulation

J.1.1 SimulationModel.java

Listing J.1: SimulationModel.java

```
1  package src;
2
3  import desmoj.*;
4  import desmoj.dist.*;
5  import desmoj.statistic.*;
6  import java.util.*;
7  import java.text.*;
8
9  public class SimulationModel extends Model {
10
11   private static int simDuration = 2000;
12   private static double simTracePeriod = 5;
13   private static double simDebugPeriod = 0;
14   //private static boolean progressbar = false;
15   private static boolean progressbar = true;
16
17
18   public static double numberUserSessions = 1000;
19   public static double totalThinkTime = 515;
20   public static double heapSize = 350;
21   public static int numRequestClasses = 8;
22   public static int numTotalRequest = 18;  // Total number of requests in workmix
                (1+1+3+3+3+3+3+1)
23
24   private desmoj.dist.RealDistExponential RequestInterArrivalTime;
25   private desmoj.dist.RealDistExponential webCpuServiceTime;
26   private desmoj.dist.RealDistExponential appCpuServiceTime;
27   private desmoj.dist.RealDistExponential dbCpuServiceTime;
28   private desmoj.dist.RealDistExponential userThinkTime;
29
```

```
30    Res lan;
31    Res webCpu;
32    Res appCpu;
33    Res dbCpu;
34
35    Accumulate webCpuAccumulate;
36    Accumulate appCpuAccumulate;
37    Accumulate dbCpuAccumulate;
38
39    public SimulationModel(Model owner, String modelName, boolean showInReport, boolean showInTrace
                ) {
40      super(owner, modelName, showInReport, showInTrace);
41    }
42
43    public String description() {
44      return "";
45    }
46
47    public double getRequestInterArrivalTime() {
48      return RequestInterArrivalTime.sample();
49    }
50    public double getWebCpuServiceTime() {
51      return webCpuServiceTime.sample();
52    }
53    public double getAppCpuServiceTime() {
54      return appCpuServiceTime.sample();
55    }
56    public double getDbCpuServiceTime() {
57      return dbCpuServiceTime.sample();
58    }
59    public double getUserThinkTime() {
60      return userThinkTime.sample();
61    }
62
63
64    public void doInitialSchedules() {
65      for (int i=1; i<=numRequestClasses; i++) {
66        // Start a Source (request generator) for each request class
67        Source newSource = new Source(this,"RequestGenerator"+i,false);
68        newSource.schedule(new SimTime(this.getUserThinkTime()));
69      }
70    }
71
72    public void init() {
73
74      RequestInterArrivalTime = new RealDistExponential(this,"requestInterArrivalTime", 1, true,
                  false);
75      RequestInterArrivalTime.setNonNegative(true);
76      RequestInterArrivalTime.setSeed(435134134);
77
78      webCpuServiceTime = new RealDistExponential(this,"WebCpuServiceTimeStream", 1, true, false);
79      webCpuServiceTime.setNonNegative(true);
80      webCpuServiceTime.setSeed(53214352);
81
82      appCpuServiceTime = new RealDistExponential(this,"AppCpuServiceTimeStream", 1, true, false);
83      appCpuServiceTime.setNonNegative(true);
84      appCpuServiceTime.setSeed(572446315);
85
86      dbCpuServiceTime = new RealDistExponential(this,"DbCpuServiceTimeStream", 1, true, false);
87      dbCpuServiceTime.setNonNegative(true);
88      dbCpuServiceTime.setSeed(23467543);
89
90      userThinkTime = new RealDistExponential(this,"userThinkTimeStream", 1, true, false);
91      userThinkTime.setNonNegative(true);
92      userThinkTime.setSeed(57245614);
93
94      webCpu  = new Res(this, "WebCpu", 1, true, true);
95      appCpu  = new Res(this, "AppCpu", 1, true, true);
96      dbCpu   = new Res(this, "DatabaseCpu", 1, true, true);
97    }
98
99    public static void main(java.lang.String[] args) {
```

```
100     // print time
101     Date currentTime = new Date();
102     DateFormat sdf = new SimpleDateFormat();
103     System.out.println("Experiment started at: " + sdf.format(currentTime));
104 
105     Experiment Simulation_Experiment = new Experiment("BankApp_Experiment");
106     SimulationModel simModel = new SimulationModel(null, "SimulationModel", true, false);
107     simModel.connectToExperiment(Simulation_Experiment);
108 
109     Simulation_Experiment.setShowProgressBar(progressbar);
110     Simulation_Experiment.tracePeriod(new SimTime(0.0),new SimTime(simTracePeriod) );
111     Simulation_Experiment.debugPeriod(new SimTime(0.0),new SimTime(simDebugPeriod) );
112     Simulation_Experiment.stop(new SimTime(simDuration));
113     Simulation_Experiment.start();
114     Simulation_Experiment.report();
115     Simulation_Experiment.finish();
116 
117     // print time
118     currentTime = new Date();
119     System.out.println("Experiment ended  at: " + sdf.format(currentTime));
120   }
121 }
```

J.1.2 Source.java

Listing J.2: Source.java

```
1  package src;
2  import desmoj.*;
3  import java.io.PrintStream;
4  import java.util.Random;
5 
6  public class Source extends ExternalEvent {
7 
8    private SimulationModel myModel;
9    private int requestClass = 0;
10   private double requestInterArrivalMean = 0;
11 
12 
13   public Source(Model owner, String name, boolean showInReport) {
14     super(owner, name, showInReport);
15     myModel = (SimulationModel) owner;
16   }
17 
18   public void eventRoutine() {
19     int intensity = 0;
20 
21     // Decide which class to generate requests [1..8]
22     // Object is named f.x "RequestGenerator2 #1", where 2 is the request class
23     requestClass = Integer.parseInt(this.getName().substring(16,17));
24 
25     // Decide InterArrivalTime of new request
26     // Specifying how often a request arrives at the system (relatively to the other request
                 classes)
27     switch (requestClass) {
28     case 1: intensity = 1;break;
29     case 2: intensity = 1;break;
30     case 3: intensity = 3;break;
31     case 4: intensity = 3;break;
32     case 5: intensity = 3;break;
33     case 6: intensity = 3;break;
34     case 7: intensity = 3;break;
35     case 8: intensity = 1;break;
36     }
37 
38     // Calculate IAT pr request.
39     requestInterArrivalMean = SimulationModel.totalThinkTime / ( intensity * SimulationModel.
                 numberUserSessions );
40 
41     Request newRequest = new Request(myModel, "Request"+requestClass, true);
42     newRequest.activateAfter(this);
```

```
43      this.schedule(new SimTime( requestInterArrivalMean * myModel.getRequestInterArrivalTime()));
44    }
45
46  }
```

J.1.3 Request.java

Listing J.3: Request.java

```
1   package src;
2
3   import desmoj.*;
4   import desmoj.dist.*;
5
6   public class Request extends SimProcess {
7
8     private SimulationModel myModel;
9     private int requestClass;
10
11    // PARAMETER_UNIT=1 for seconds, 0.001 for milliseconds;
12    private static double PARAMETER_UNIT = 0.001;
13
14    private void visitResource(Res resource, double holdTime, int slots) {
15      resource.provide(slots);
16      hold(new SimTime(holdTime));
17      resource.takeBack(slots);
18    }
19
20    public Request(Model owner, String name, boolean showInTrace) {
21      super(owner, name, showInTrace);
22      myModel = (SimulationModel) owner;
23    }
24
25    public void lifeCycle() {
26      double webCpuTime, appCpuTime, dbCpuTime, thinkTime;
27      double[] parameters = { 0, 0, 0 };
28      double webExtraServiceDemand = 0;
29
30      // Object is named f.x "Request2 #534", where 2 is the request type
31      requestClass = Integer.parseInt(this.getName().substring(7,8));
32
33      // parameters[0] denotes web, [1] is app, and [2] is database
34      switch (requestClass) {
35      case 1:
36        parameters[0] =   8.3;
37        parameters[1] =   0.1;
38        parameters[2] =   0.0;
39        break;
40      case 2:
41        parameters[0] =  41.2;
42        parameters[1] =  17.6;
43        parameters[2] =   0.6;
44        break;
45      case 3:
46        parameters[0] =  26.1;
47        parameters[1] =   8.5;
48        parameters[2] =   0.5;
49        break;
50      case 4:
51        parameters[0] =  25.9;
52        parameters[1] =   8.1;
53        parameters[2] =   0.5;
54        break;
55      case 5:
56        parameters[0] =   8.7;
57        parameters[1] =   0.3;
58        parameters[2] =   0.1;
59        break;
60      case 6:
61        parameters[0] =   6.6;
62        parameters[1] =   0.2;
```

```
63        parameters[2] =   0.1;
64        break;
65      case 7:
66        parameters[0] =  20.5;
67        parameters[1] =   7.7;
68        parameters[2] =   3.1;
69        break;
70      case 8:
71        parameters[0] =   7.0;
72        parameters[1] =   0.6;
73        parameters[2] =   0.1;
74        break;
75      }
76
77      // Add extra service demand from complexity functions
78      // Function produces extra service demand for all requests in the user session, so divide it
                  amongst all requests
79      webExtraServiceDemand = (-0.007 * SimulationModel.heapSize + 11.7) / SimulationModel.
                  numTotalRequest;   //web garbage collector complexity function
80
81      parameters[0] += webExtraServiceDemand;
82
83      //System.out.println("Request class: " + requestClass + ":  " + parameters[0] + ", " +
                  parameters[1] + ", " + parameters[2]);
84      // Web CPU
85      webCpuTime = ( parameters[0] * myModel.getWebCpuServiceTime() ) * PARAMETER_UNIT;
86      visitResource(myModel.webCpu, webCpuTime, 1);
87
88      // Application CPU
89      appCpuTime = ( parameters[1] * myModel.getWebCpuServiceTime() ) * PARAMETER_UNIT;
90      visitResource(myModel.appCpu, appCpuTime, 1);
91
92      // DB CPU
93      dbCpuTime = ( parameters[2] * myModel.getWebCpuServiceTime() ) * PARAMETER_UNIT;
94      visitResource(myModel.dbCpu, dbCpuTime, 1);
95    }
96  }
```