

Performance Modeling of Object Middleware

Marcel Harkema

September 25, 2011 Draft

CONTENTS

1	Introduction	1
1.1	The emergence of Internet e-business applications	1
1.2	Objectives and scope	4
1.3	Outline of the thesis	6
2	Performance Measurement	9
2.1	Performance measurement activities	9
2.2	Measurement terminology and concepts	12
2.3	Measurement APIs and tools	16
2.4	Summary	30
3	The Java Performance Monitoring Tool	31
3.1	Requirements	32
3.2	Architecture	34
3.3	Usage	35
3.4	Implementation	40
3.5	Intrusion	50
3.6	Summary	50
4	Performance Modeling of CORBA Object Middleware	53
4.1	CORBA object middleware	53
4.2	Specification and implementation of CORBA threading	56
4.3	Performance models of threading strategies	61
4.4	Workload generation	73
4.5	Throughput comparison of the threading strategies	75
4.6	Impact of marshaling	81
4.7	Modeling the thread scheduling	84
4.8	Summary	87
5	Performance Model Validation	89

5.1	Performance model implementation	89
5.2	The Distributed Applications Performance Simulator	90
5.3	Validation of the thread-pool strategy for an increasing number of dispatchers	92
5.4	Validation of the threading strategies for an increasing number of clients	98
5.5	Summary	116
6	Performance Modeling of an Interactive Web-Browsing Application	121
6.1	Interactive web-browsing applications	121
6.2	The local weather service application	122
6.3	Performance model	124
6.4	Experiments	128
6.5	Validation	132
6.6	Summary	134
7	Conclusions	137
7.1	Review of the thesis objectives	137
7.2	Future work	139
	References	141

INTRODUCTION

This chapter presents the background, the problem description, the objectives and scope, and the organization of this thesis.

1.1 The emergence of Internet e-business applications

The tremendous growth of the Internet [42] and the ongoing developments in the hardware and software industry have boosted the development of Information and Communication Technology (ICT) systems. These systems consist of geographically distributed components communicating with each other using networking technology. Such systems are commonly referred to as *distributed systems*.

A key challenge of distributed systems is interoperability: the vast diversity in hardware, operating systems, and programming languages, makes it difficult to build distributed applications. Over the past decade there have been a lot of advances in middleware technology aimed at solving this interoperability problem. *Middleware* is software that hides architectural and implementation details of an underlying system and offers well-defined interfaces instead. Some of the key advances in middleware include OMG CORBA object middleware and the Sun Java infrastructure middleware.

The Common Object Request Broker Architecture (CORBA) [41] [61] is a standard developed by the Object Management Group (OMG) [54], an inter-

national consortium of companies and institutions. OMG CORBA specifies how computational objects in a distributed and heterogeneous environment can interact with each other, regardless of which operating system and programming languages these objects run on. For instance, using CORBA object middleware, a piece of software written in the C programming language and running on a UNIX system can interact with another piece of software written in COBOL programming running on another computer system. CORBA essentially encapsulates objects that may be implemented in a wide variety of programming languages and enables them to interact with each other.

Over the past decade Java [2] has evolved into a mature programming platform. Java, developed by Sun Microsystems, hides the heterogeneity of operating systems and hardware by providing a virtual machine, i.e. it can be viewed as host infrastructure middleware. The virtual machine [50] can be programmed using the Java programming language. The Java language is based on Objective-C and Smalltalk object-oriented programming languages. In the early days of Java, it was mostly used to make platform-independent applications and so-called applets, small applications that run inside a web-page. Over the years, Java matured into a platform for building multi-tiered enterprise applications.

Today the Java Platform, Enterprise Edition (Java EE or JEE), is the *de-facto* standard for building such enterprise applications. Some of the key components of the JEE platform include the JDBC API for accessing SQL databases (allowing developers to program to a common API instead of vendor-specific APIs), technology for building interactive web-applications (the Servlet API and Java Server Pages), APIs for interpreting and manipulating XML, the RMI API for performing remote method invocations, and Enterprise JavaBeans (EJB), which is a component model for building enterprise applications. The Java platform also includes a pluggable CORBA implementation. Vendors can swap the default ORB implementation with their own. The EJB component model is built on top some of the CORBA technologies: the Java Transaction Service (JTS) is a Java binding of the CORBA Object Transaction Service (OTS), the Java Naming Service is based on the Cos Naming Service, and interoperability between EJB beans is based on CORBA's IIOP (also, CORBA clients can invoke enterprise Java beans).

The developments described above have led to the emergence of a wide variety of e-business applications, such as online ticket reservation, online banking, and online purchasing of consumer products. In the competitive market of e-businesses, a critical success factor for e-business applications is

the Quality of Service (QoS) of these applications provided to the customers [53]. The QoS includes metrics such as response time, throughput, availability, and security (e.g., credit-card payment transactions, privacy of customer data) characteristics. QoS problems may lead to customer dissatisfaction and eventually to loss of revenue. So there is a need to understand and control the end-to-end performance of these e-business applications. The end-to-end performance is a highly complex interplay between the network infrastructure, operating systems, middleware, application software, and the number of customers using the application, amongst others.

To assess the performance of their e-business applications, companies usually perform a variety of activities: (1) performance lab testing, (2) performance monitoring, and (3) performance tuning. *Performance lab testing* involves the execution of load and stress tests on applications. *Load tests* test an application under a load similar to the expected load in the production system. *Stress tests* are used to test the stability and performance of the system under a load much higher than the expected load. Although lab-testing efforts are undoubtedly useful, there are two major disadvantages. First, building a production-like lab environment may be very costly, and second, performing load and stress tests and interpreting the results are usually very time consuming, and hence highly expensive. *Performance monitoring* is usually performed to keep track of high-level performance metrics such as service availability and end-to-end response times, but also to keep track of the consumption of low-level system resources, such as CPU utilization and network bandwidth consumption. Results from lab testing and performance monitoring provide input for *tuning* the performance of an application.

A common drawback of the aforementioned performance assessment activities is that their ability to predict the performance under projected growth of the workload in order to timely anticipate on performance degradation (e.g., by planning system upgrades or architectural modifications) is limited. This raises the need to *complement* these activities with methods specifically developed for *performance prediction* [59]. To this end, various modeling and analysis techniques have been developed over the past few decades, see, e.g., [3], [32], [36], [45], [56], [66], and references therein. These *performance models* are abstractions of the real system describing the parts of the system that are relevant to performance. Such a performance model typically contains information on the architecture of the system, the physical resources (CPU, memory, disk, and network) as well as logical resources (threads, locks, etc.) in the system, and the workload of the system [65], which consists of a statistical description of the arriving requests and resource usage by those

requests. In the design phase of an application, performance models can be used to evaluate different design alternatives. In the production phase of an application, performance models can be used to predict the performance under projected growth of the workload, so that performance degradation can be timely anticipated. The performance models can be evaluated, using simulation, analytical methods, or numerical approximations to obtain performance measures, such as utilization of resources (useful to find bottleneck resources), throughputs, response times, and their statistical distributions.

In order to be useful for performance prediction, a performance model needs to predict the performance of the modeled system accurately. The ultimate validation of the performance model is to compare them with real-world, or test-bed, results. The ‘art’ of performance modeling is to develop models that only include as few components as possible, while still accurately (enough) predicting the performance of the modeled system. Often there is a trade-off between the complexity of the performance model and the required accuracy of the predictions.

Software is increasingly becoming complex [17]. Today’s e-business applications are multi-tiered systems comprising of a mix of databases, middleware, web servers, application servers, application frameworks, and business logic. Often little is known about the inner workings and performance of these servers, software components, and frameworks. With this increasing complexity of software [52] and the observation that the capacity of networking resources is growing faster than the capacity of processor resources [9], performance modeling of this software is of an increasing importance.

1.2 Objectives and scope

The overall objective of this thesis is to develop and validate quantitative performance models of distributed applications based on middleware technology. We limit the scope of our research to OMG CORBA object middleware and the Java EE platform.

In order to be able to model the performance of software, insight in its execution behavior is needed. This raises questions such as:

- What are the use-cases for the application that need modeling?
- Into which pieces can the response time for a specific use-case be broken down? Which part of the response time can be attributed to the business logic, 3rd party frameworks and components, the database

server, the middleware layer, the Java virtual machine, and the operating system?

- Which of these parts need to be present in the performance model?
- What logical resources (i.e. threads and locks) does the application have?
- What logical resources and physical resources (e.g., CPU cycles, network bandwidth) are needed for each use-case?

These questions in turn raise another question: how do we obtain this information? Documentation, e.g., standard specifications, design documentation of the application in the form of UML diagrams, and source code annotations, is an important source of information. However, in many cases documentation lacks detail or is not available. Also, documentation does not provide a full picture of the application, answering every performance question one might have. In these cases it is needed to monitor execution behavior and measure performance in a running system. This is quite a challenge considering the complexity of enterprise applications.

We have split the overall objective of this thesis into the following set of sub-objectives:

1. Investigate and develop techniques to identify and quantify performance aspects of Java applications and components. These techniques will enable us to learn about performance aspects of software, and to quantify these performance aspects.
2. Obtain insight in the performance aspects of the Java virtual machine.
3. Obtain insight in the performance aspects of CORBA object middleware.
4. Obtain insight in the impact of multi-threading and the influence of the operating system's thread scheduler on the performance of threaded applications.
5. Combine these insights to construct quantitative performance models for CORBA object middleware.
6. Validate these performance models by comparing model-based results with real-world measurements.

1.3 Outline of the thesis

The different chapters of this thesis are organized as follows:

Chapter 1, titled 'Introduction', presents the background, problem description, objectives and scope of this thesis.

Chapter 2, titled 'Performance Measurement', introduces performance concepts, performance measurement activities, terminology and concepts, discusses measurement difficulties, and provides an overview of techniques, APIs and tools for performance measurement in applications, their supporting software layers, and hardware.

Chapter 3, titled 'Java Performance Monitoring Tool', presents our performance monitoring tool for Java applications and the Java host infrastructure middleware layer.

Chapter 4, titled 'Performance Modeling of CORBA Object Middleware', discusses the inner workings of CORBA object middleware and presents our performance models for CORBA object middleware.

Chapter 5, titled 'Performance Model Validation', describes simulations implementing the performance models along with their experimental validation.

Chapter 6, titled 'Performance Modeling of an Interactive Web-Browsing Application', describes a performance model of an interactive web-application.

Chapter 7, titled 'Conclusions', presents the conclusions of this thesis, evaluates how the thesis objectives have been achieved and gives directions for further research.

These chapters are based on the following publications: [25], [18], [27], [28], [26], [23], [24], [19], [21], [22], [20] and [60]. The research was mainly done during the period 2001-2006.

The following figure, Figure 1.1, illustrates our approach to reaching the overall objective of validated quantitative performance models of applications based on middleware technology.

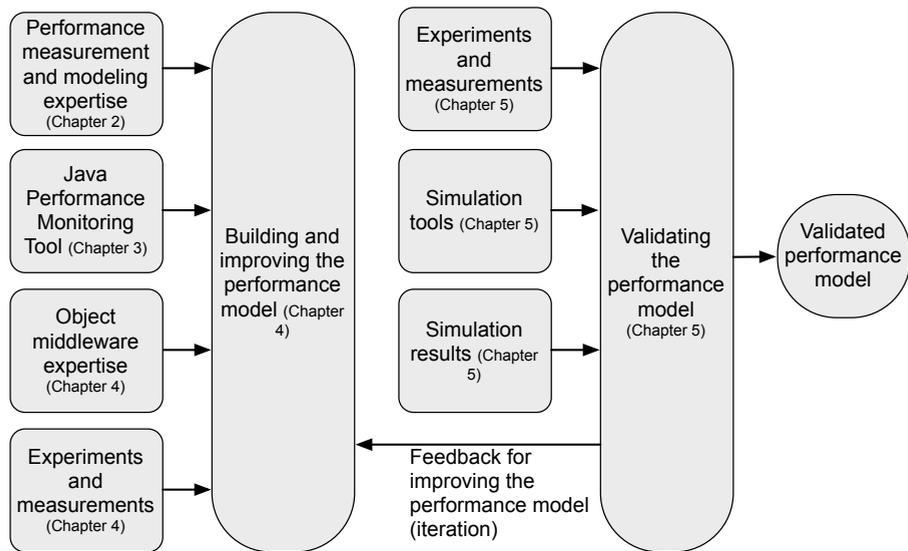


Figure 1.1: Performance modeling cycle

PERFORMANCE MEASUREMENT

In this chapter we introduce software performance monitoring and measurement concepts and discuss the various monitoring and measurement facilities available at the different layers of Java based distributed applications, from the application layer, the middleware layers, to the operating system and network layers.

This chapter is structured as follows. Section 2.1 is an overview of performance measurement activities. Section 2.2 introduces performance measurement terminology and concepts. Section 2.3 presents an overview of performance measurement APIs and tools. Section 2.4 summarizes this chapter.

2.1 Performance measurement activities

A wide variety of performance measurement activities are available, each targeted to answer specific performance related questions. In this section we give a brief overview of these activities.

2.1.1 Benchmarking

Benchmarking is a performance measurement activity that uses some standardized tests to compare the performance of alternative computer systems, components, or applications. The benchmark results should be indicative of the performance of the system or application in the real-world, therefore it is

important that the workload executed by the benchmark is representative of the real-world workload. Typical benchmark performance measures are response times for single operations and maximum rates for operations.

Benchmarks are used for various reasons. One of the most common reasons is to compare performance of various hardware or software procurement alternatives. Benchmarks are also useful as a diagnostic tool, comparing performance of some system against a well-known system, so that performance problems can be pinpointed. While benchmark results can give some insight in a system, the results do not provide a complete explanation of the inner working of a system. Therefore benchmarking does not yield enough information to develop performance models of systems.

Various standardization bodies exist for benchmarking. Among the most well-known are BAPco (Business Applications Performance Corporation) who develop a set of benchmarks to evaluate the performance of personal computers running popular software applications and operating systems. BAPco's SYSmark benchmark evaluates the performance of a system from a business client point of view, running a workload on the system that represents office productivity activities, for instance word-processing or spreadsheet usage. SPEC (System Performance Evaluation Corporation) defines a wide variety of benchmarks for CPUs (SPEC CPU2006), JEE application servers (SPECjEnterprise2010), Java business applications (SPECjbb2005), client-side Java virtual machines (SPECjvm2008), and web servers (SPECweb2005), among others. TPC (Transaction Processing Council) defines industry benchmarks for transaction processing, databases, and e-commerce servers. Among others, the TPC benchmark suite includes the TPC-W benchmark, which measures the performance of business oriented transactional web servers in transactions per second and the TPC-C and TPC-H benchmarks that measure performance of database management systems (DBMS) transactions. SPC (Storage Performance Council) defines benchmarks for characterizing the performance of storage systems, e.g. enterprise storage area networks (SANs).

2.1.2 Performance testing

The objective of performance testing is to understand how systems behave under specific workload scenarios. Contrary to benchmarking, which is used to evaluate common-off-the-shelf software and hardware, performance testing can be tailored to a specific system, application, and workload.

Various kinds of workload can be used with performance testing, investing

specific performance questions. For instance, a steady-state statistical workload can be used, representing the usual workload on the system. This is often referred to as *load testing*. This workload can be gradually increased to find the maximum workload under which the system is still stable. This is referred to as *maximum sustainable load testing*, e.g. [55]. *Stress testing* is used to investigate system behavior under deliberately constant heavy workload. Stress testing can uncover bugs in the system and performance bottlenecks. Finally, *spike or burst testing* refers to testing system behavior under a temporary high load, for instance a sudden increase in users [1]. Again, this is used to find bugs, performance bottlenecks, and to test system stability during a temporary heavy load.

Performance testing can provide a wealth of performance information, answering specific questions a performance modeler may have regarding the impact of specific workloads on the performance behavior of a system. However, the externally observable measures (*what* happened), such as system response time, throughput and resource utilization, usually provided by performance tests need to be accompanied by more in-depth performance information on the internal performance behavior of the system, explaining the externally observed performance behavior (*why* it happened). Performance monitoring tools, such as profilers or tracers, can be used to investigate the internal performance behavior.

2.1.3 Performance monitoring

Performance monitoring [51] refers to a wide range of techniques and tools to observe, and sometimes record, the performance behavior of a system, or part of a system.

Performance monitoring comes in many different flavors, ranging from observing end-to-end performance behavior to observing cache misses [64]. Performance monitors have many uses [48], including analyzing performance problems uncovered by performance testing, collecting performance information for performance modelers, gathering performance data for load balancing decisions, and monitoring whether service level agreements (SLAs) are met [10].

The remainder of this chapter discusses performance measurement and monitoring techniques and tools, and their terminology.

2.2 Measurement terminology and concepts

In this section we present measurement terminology and concepts.

2.2.1 System-level and program-level measurement

Two categories of performance measurement data can be distinguished: *system-level* measurements and *program-level* measurements [32]. System-level measurements represent global system performance information, such as CPU utilization, number of page faults, free memory, etc. Program-level measurements are specific to some application running in the system, such as the portion of CPU time used by the particular application, used memory, page faults caused by the application, etc.

2.2.2 Black-box and white-box observations

The performance of a computer system or application can be evaluated from an external and internal perspective. So-called black-box performance measurements measure externally observable performance metrics, like response times, throughput, and global resource utilization (e.g. CPU utilization of the whole system).

White-box performance measurements are done ‘inside’ the system or application under study, often using specialized tools such as monitors described below.

Figure 2.1 illustrates black-box and white-box observations.

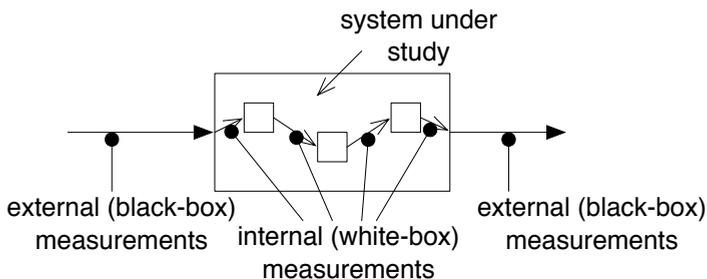


Figure 2.1: Black-box and white-box observation

2.2.3 Monitoring

A monitor is a piece of software, hardware, or a hybrid (mix) [32], that extracts dynamic information concerning a computational process, as that process executes [49]. Monitoring can be targeted to various classes of functionality [48], including correctness checking, security, debugging and testing, and on-line steering. This thesis focuses on monitoring for performance evaluation and program understanding purposes.

Monitoring consists of the following activities [49] [32]:

- *Preparation.* During preparation, the first step is to decide what kind of information monitoring should collect from the program. For instance, performance data regarding disk operations can be collected or the number and kind of CPU instructions the program needs. The second preparation step is to determine where to collect this information. Monitoring tools often specialize in some part of the system where they collect performance information and the kind of performance information they collect.
- *Data collection.* After preparing the monitoring we can execute the process. During execution of the process, the monitor observes this process and collects the performance information.
- *Data processing.* This activity involves interpretation, transformation, checking, analyses, and testing of the collected performance data.
- *Presentation of the performance data.* Presentation involves reporting the performance data to the user of the monitor.

2.2.4 State sampling and event-driven monitoring

In general, two types of monitors can be distinguished: time-driven monitors and event-driven monitors [32].

Time-driven monitoring observes the state of the monitored process at certain time intervals. This approach, also known as sampling, state-based monitoring, and clock-driven monitoring, is often used to determine performance bottlenecks in software. For instance, by observing a machine's call-stack every X milliseconds, a list of the most frequently used software routines (called hot spots) and routines using large amounts of processing times can be obtained. Time-driven monitoring does not provide complete behavioral information, only snapshots.

Event-driven monitoring is a monitoring technique where events in the system are observed. An event represents a unit of behavior, e.g., the creation of a new thread in the system and the invocation of a method on an object. When besides the occurrence of the event itself (what occurred), a portion of the system state is recorded that uniquely identifies an event [39], such as timing information (*when* did the event occur) and location information (*where* exactly did it occur, for instance a particular software routine or computational object), we refer to this as tracing. Events have temporal and causal relationships. Temporal relationships between events reflect the ordering of those events according to some clock, which could be the system's physical clock (if all events occur in the same system), or some logical clock (when monitoring distributed systems) [34]. Causal relationships between events reflect cause and effect between events, for instance accessing some data structure may result in a page fault event if the data is not present in the physical memory, but swapped out to disk. As we will see later on, causal relations between events are not always evident. In some cases the monitor needs to record extra information with the events to allow event correlation during event trace data processing.

2.2.5 Online and offline monitoring

Online and offline monitors differ in the moment when data processing and presentation of the data takes place. In traditional offline monitoring tools the preparation activity takes place before execution of the monitored system, the data collection activity takes place during execution, the data processing and presentation activities take place after execution.

In online monitoring systems [48] the data processing occurs during execution time. Example application areas of online monitoring systems are security, so security violations can be detected as they occur, and performance control, where monitoring data is used to constantly adapt the configuration of a system to meet performance goals. Online monitoring systems may also present monitoring data to the user, for instance actual security and performance monitoring results may be reported in a monitoring console.

2.2.6 Instrumentation

Monitoring requires functionality in the system to collect the monitoring data. The process of inserting the required functionality for monitoring in the system is called *instrumentation*.

There are various options of instrumenting the system to collect monitoring data for an application running on the system. We can instrument the application itself, this is called *direct instrumentation*, or we can instrument the environment in which the application runs, this is referred to as *indirect instrumentation*. The environment includes the operating system, libraries the application uses, virtual machines, and such. Below we list often used direct and indirect instrumentation techniques:

Modification of the application's source code. This can be done in various ways. First, instrumentation can manually be inserted in the source code before compilation time. Depending on the amount of monitoring information needed, this can be a quite labor intensive job. Instead of manual instrumentation more automated ways of instrumentation can be used to add instrumentation to the source code. A source code pre-processor can be used to automatically insert instrumentation in the source code before actually compiling the source code. The instrumentation process could be based on some configuration file containing information on where to insert instrumentation in the source code. Using a pre-processor to insert instrumentation code has the advantage that it is easier to change the instrumentation (e.g., because other monitoring data is needed); all that needs to be done is change the configuration file, run the pre-processor and re-compile the application. Similar to using a pre-processor to insert instrumentation, the compiler itself can be altered to insert instrumentation as it compiles the source code into binary code.

Modification of the application's binary code. Instead of inserting the instrumentation at compile time, described above, we can also insert instrumentation just before run time. Binary instrumentation is quite difficult, since binary code is much harder to interpret than source code. The advantages are that it does not require the source code to be available, and that re-compilation of the application is not needed to insert or alter the instrumentation.

Using vendor supplied APIs. Server applications such as database servers, web servers, and application servers often include programming interfaces or other access points to monitoring information.

Monitoring the software environment. The above techniques are direct instrumentation techniques. Sometimes we cannot directly instrument the application. We can then monitor the environment of the application, such as libraries, runtime systems (virtual machines), and the operating system. A disadvantage of indirect instrumentation is that we will not be able to ob-

serve events inside the application, only interactions with the environment can be observed. The advantage is that instrumentation is not application specific, instead it is more generic.

Monitoring the hardware environment. Another indirect instrumentation technique is using hardware monitoring information. Even more so than monitoring the software environment, it is hard to correlate this monitoring information to activity in the application we want to collect monitoring data for.

Note that a monitoring solution may combine several of the above techniques to observe an application. For instance, application events obtained by instrumenting the source code may be combined with monitoring information provided by hardware and events occurring in the operating system kernel, such as thread context switches.

2.2.7 Overhead, intrusion and perturbation

Adding monitoring instrumentation to a system causes perturbations in the system. This interference in the normal processing of a system is referred to as *intrusion*.

Software instrumentation requires the use of system resources, such as the CPU, threads, and memory, which may also be used by the monitored application. This may cause the application to perform worse than the un-instrumented version of the application. The difference in performance between the instrumented and un-instrumented application is called *performance overhead*. Besides perturbing the performance of a system, instrumentation can also change the execution behavior of a system. For instance, CPU cycle consumption of the instrumentation and processing threads belonging to the monitoring tool may change the thread scheduling behavior of the application.

There is also non-execution related intrusion, such as replacing an application with an instrumented application, changing the system's configuration and deployment to facilitate monitoring, and requiring an application to be restarted after instrumentation is added.

2.3 Measurement APIs and tools

In this section we discuss APIs and tools suitable for performance measurement on the UNIX and Windows operating systems, and in the Java

environment.

2.3.1 High-resolution timing and hardware counters

Performance measurement of software applications requires high resolution timestamps. Many operating systems, including Windows and Linux, use the system's clock interrupt to drive the operating system clock. The frequency of the clock interrupt then determines the resolution of the clock. On x86 based systems clock interrupts are historically configured to occur every 10 milliseconds, though with the advent of more powerful processors 1 millisecond intervals are becoming common too (recent versions of the Linux kernel offer configurable timer interrupt intervals). A higher frequency will result in more interrupt overhead. For measurement of activity within software applications 10 milliseconds resolution is too coarse grained.

Modern processors, such as the Intel x86 family since the Pentium series, have performance counters embedded in the processor. One of these counters is a timestamp counter (TSC) which is increased every processor cycle. Timestamps can be calculated by dividing the timestamp counter by the processor frequency. On processors targeted at the mobile market, such as the Intel Pentium M family, the timestamp counters are not incremented at a constant rate, since the processor frequency can be varied depending on the system's workload and power saving requirements. On these systems an average processor frequency can be used to calculate timestamps, with loss of accuracy. Other events that can be counted are retired instructions, cache misses, and interactions with the bus.

Table 2.1 lists some options for high-resolution timing.

For performance modeling purposes we are interested in the consumption of CPU resources. By using hardware counters we can measure the number of processor cycles and the number of retired instructions. However, these counters are global, i.e. for all running processes, while we are interested in event counts related to the processes that are part of the software we are monitoring. Per-process (or per-thread) monitoring of hardware event counters requires instrumentation of the context switch routine in the operating system's kernel. The 'perfctr' kernel patch for Linux on the x86 platform implements such per-thread monitoring of hardware event counters (called virtual counters in perfctr). Similar hardware counter monitoring packages are available for other platforms and operating systems, such as 'perfmon' for Linux on the Intel Itanium and PPC-64 platforms (integrated in the Linux 2.6 kernel) and the 'pctx' library on Sun Solaris on the Sun Sparc platform.

Method	System	Measurement type
gettimeofday(2)	UNIX	Wall-clock time in micro-seconds. Accuracy varies, on older systems it can be in the order of tenths of microseconds, on modern systems it is 1 microsecond. Modern UNIX variants based the time on hardware cycle counters. E.g., in Linux the TSC is used on Intel x86 based machines.
gethrtime(3c)	Sun Solaris and some real-time UNIX variants	Wall-clock time in nanoseconds. Accuracy is in the tenths of nanoseconds, depending on the processor. The time is based on a hardware cycle counter.
gethrvtime(3c)	Sun Solaris and some real-time UNIX variants	Variant of gethrtime(3c). Per light-weight-process (LWP) CPU time in nanoseconds.
QueryPerformanceCounter and QueryPerformanceFrequency	Microsoft Windows	High resolution timestamps based on hardware cycle counters. The QueryPerformanceCounter function returns the number of cycles. The QueryPerformanceFrequency returns the frequency of the counter. Accuracy is around a couple of microseconds on modern hardware.

Table 2.1: Various high-resolution timestamp functions

Which hardware counters are available and how they can be accessed differs per processor type and operating system. This makes it difficult to create portable performance measurement routines. Libraries, such as PAPI [7] and PCL [5], offer standardized APIs to access hardware counters.

2.3.2 Information provided by the operating system

Many operating systems keep performance information that can be accessed by users.

Global and process-level performance information

The process information pseudo file-system ‘proc-fs’, available in some UNIX variants (e.g. Linux) is a special-purpose virtual file-system, where kernel state information, including performance related information, is mapped into memory. The file-system is mounted at /proc. Proc-fs stores global performance measures, such as CPU consumption information, disk access information, memory usage information, and network information. It also stores non performance related information, such as drivers loaded, hardware connected to the USB bus, and disk geometry information. Some files in the proc file-system can be modified by the (root) user, changing parameters in the operating system kernel, for instance various TCP/IP networking options can be configured. The proc file-system also stores per-process

information, such as CPU consumption for each process and memory consumption for each process. The files in the `proc-fs` filesystem usually are text files which have to be parsed by the user. Another UNIX variant, Solaris, also maintains kernel state information, but offers a different access mechanism: the kernel statistics facility 'kstat'. User applications can access the kstat facility by linking with the `libkstat` C library.

Microsoft Windows also offers access to performance information of the operating system, through the Windows registry API. The Windows registry is a hierarchical database used to store settings of applications and the operating system. The performance data can be accessed using the `HKEY_PERFORMANCE_DATA` registry key. The performance data is not actually stored in the registry (i.e. stored on disk), instead accessing performance data using the registry API will cause the API to call operating system and application provided handlers to obtain the information. Windows also offers the Performance Data Helper (PDH) library, which hides many of the complexities of the registry API.

The offered performance data by the registry is similar to the data offered by the `proc-fs` and `kstat` performance interfaces described above.

Performance measurement and monitoring applications can use data from these operating system supplied performance data repositories. Usually operating systems offer ready to use performance monitoring applications also based on these performance data repositories. Examples of such applications are 'top', a program that lists processes and their performance information such as CPU and memory consumption, and the Windows Task Manager and Windows Performance Monitor applications.

Kernel event tracing

The above APIs provide the user with global and per-process performance counters, such as the global CPU utilization, amount of CPU time consumed by a process, and the number of disk access by a process. For a more detailed view on a system's performance kernel event tracing can be used. Kernel event tracing allows the user to subscribe to events of interest in the kernel, such as thread context switches, opening files, and sending data on the network. So, instead of just counting disk accesses, kernel event tracing informs the user of a disk access as it occurs together with context information such as the process ID under which the disk access event occurs and the time of the event. This provides the user with more detailed information. However, event tracing is more intrusive than event counting. Kernel event tracing facil-

ities are less common than facilities offering global and per-process counters. Recently, Microsoft introduced the Event Tracing for Windows (ETW) subsystem [40] in Windows 2000 and Windows XP. On Linux, the Linux Trace Toolkit (LTT) [67] is available, but not integrated yet in the production kernel. In Sun Solaris version 10 the DTrace [8] facility was added.

2.3.3 The application layer

Applications may provide performance monitoring facilities in the form of APIs or log-files. For instance, server applications, such as database servers, web servers, and application servers, often include programming interfaces or other access points to monitoring information. For instance, the Apache web server provides a module which can be loaded into the web server that provides various kinds of information, such as the CPU load, number of idle and busy servers, and server throughput. Most web servers are also able to log requests in log-files, which can be processed by the user to gather all kinds of statistics, such as frequently requested pages. Another example is the MySQL database server that can provide a list of running server threads, what queries they are processing, contended database table locks, and such.

2.3.4 The Java infrastructure middleware layer

Over the past years Java has evolved into a mature programming platform. Java's portability and ease of programming makes it a popular choice for implementing enterprise applications and off-the-shelf components such as middleware.

Java is an object-oriented programming language based on Smalltalk, and Objective-C. Unlike Smalltalk and Objective-C it uses static type checking. Java source-code is compiled to byte-code which can be interpreted by the Java virtual machine, although there are compilers that directly compile Java source-code to native machine code (e.g. GNU GCJ). The Java virtual machine is a runtime system providing a platform independent way of executing byte-code on many different architectures and operating systems. This makes Java a host infrastructure middleware, sitting between the operating system (and system libraries) and the applications running on top of the Java virtual machine. Java applications are shielded from operating system and computer hardware architectures underneath the virtual machine.

Java is multi-threaded, in most virtual machines the threads are mapped to light-weight operating system processes/threads. Monitors [30] are used as the underlying synchronization mechanism to implement mutual exclusion

and cooperation between threads. In Java objects that are allocated and no longer used (dead) are garbage collected. There are simple facilities to make object release explicit, but it's not common to use them. While garbage collection is useful to programmers (no need to worry about releasing allocated memory manually, and no memory leaks), it can lead to careless programming practices, stressing the garbage collector a lot (wasting a lot of CPU cycles).

Java's inner workings are described in detail by the Java Virtual Machine Specification [37] and the Java Language Specification [16].

The completion time of Java method invocations depends on many factors:

- CPU cycles used by the application code.
- Sharing of the CPU(s) by multiple threads.
- Time spent waiting for resources to become available (e.g., contention Java monitors). The more threads share the same resources, the higher the contention for these resources. Obviously, the duration of critical sections is also a factor that determines contention.
- Disk I/O and network I/O.
- Latencies incurred using software outside the virtual machine. This includes accessing remote databases, remote method invocation on Java objects in other virtual machines, etc.
- CPU cycles used by the Java virtual machine and other supporting software, such as system libraries and the operating system.
- Garbage collection. By default a stop-the-world garbage collection using copying (for younger objects) and mark-and-sweep (for older objects) is used in Sun's Java virtual machine [63]. New garbage collection algorithms have been introduced in the 1.4 series, but are not enabled by default. Stop-the-world garbage collection can have significant impact on application performance, since program execution is suspended during garbage collection. Also, the large number of memory management / garbage collection parameters of the virtual machine make it difficult to find optimal settings for applications.
- Run-time compilation techniques may improve performance of so-called 'hot spots' (often invoked methods and/or methods with loops).

Performance analysts need ways to quantify the method completion times and the dependencies on these method completion times. The Java virtual machine provides a number of interfaces allowing us to observe the internal behavior of the virtual machine: the JVMDI and JVMPI. The Java Virtual Machine Debug Interface (JVMDI) is a programming interface that supports application debuggers. The JVMDI is not suited for performance measurement, but it can be used to observe control flows and state within the Java virtual machine. The Java Virtual Machine Profiler Interface (JVMPI) is a programming interface that supports application profilers. Like the JVMDI, the JVMPI also observes control flows and state within the Java virtual machine. However, the JVMDI observes qualitative behavior of the application (supporting functional debugging of an application) while the JVMPI observes quantitative behavior (supporting performance debugging of an application).

Java Virtual Machine Debug Interface (JVMDI)

The JVMDI provides core functionality needed to build debuggers and other programming tools for the Java platform. JVMDI allows the user to inspect the state of the virtual machine as well as control over the execution of applications. JVMDI provides a two-way interface which can be used to receive and subscribe to events of interest and query and control the application.

The JVMDI supports the following functionality:

- *Memory management hooks.* Functions to allocate memory and replace the default memory allocator with a custom one.
- *Thread and thread group execution functions.* Allowing the status of threads to be queried (including information on monitors), threads to be suspended, resumed, stopped (killed), or interrupted (waking up a blocked thread and sending an exception).
- *Stack frame access.* Functions to inspect the frames on call stacks of threads. Stack frames are used to store data structures needed to implement sub-routine calls, i.e. method invocation and return.
- *Local variables functions.* Functions to get and set local variables.
- *Breakpoint functions.* Functions to set and clear breakpoints in Java applications. Breakpoints trigger the debugger when a certain condition is reached, e.g. some method implementation.

- *Functions for watching fields.* Allowing the debugger to receive an event when a variable is accessed or modified in the application. Functions for obtaining class, object, method, and field information. This includes class definitions, source code information (file name of source file, line numbers), signatures of methods, defined variables, local variables in methods, etc. This mostly concerns static information allowing the application structure to be queried.
- *Raw monitor functions.* These functions provide the debugger developer with monitors needed to make the debugger functionality using the JVMDI multi-thread capable. The Java application may have more than one thread triggering debugger functionality (events) at the same time. Using the raw monitors the data structures of the debugger can be locked for a single thread while they are modified.

The JVMDI requires the virtual machine to run in debugging mode, making JVMDI less suitable for performance measurement (because of the debugging overhead) and production systems (e.g., for online performance monitoring in production systems).

JVMDI is part of the Java Platform Debugger Architecture (JPDA), but can be used independently of the other parts. Besides the JVMDI, the JPDA parts are JDWP and JDI. JDWP is a wire protocol allowing debug information to be passed between the debuggee virtual machine and the debugger front-end, which may run in another virtual machine and hence can run on another host. The JDWP even allows debugger front-ends to be written in other programming languages than Java. JDI is a high-level Java API for supporting debugger front-ends. JDI implements common functionality required by debuggers and other programming tools. The JDI is not required to write debugger and other programming tools, both the JVMDI and JDWP can be used independently from JDI.

Java Virtual Machine Profiler Interface (JVMPi)

The JVMPi allows a user provided profiler agent to observe events in the Java virtual machine. The profiler agent is a dynamically linked library written in C or C++. By subscribing to the events of interest, using JVMPi's event subscription interface, the profiler agent can collect profiling information on behalf of the monitoring tool. Figure 2.2 depicts the interactions between the JVMPi and the profiler agent.

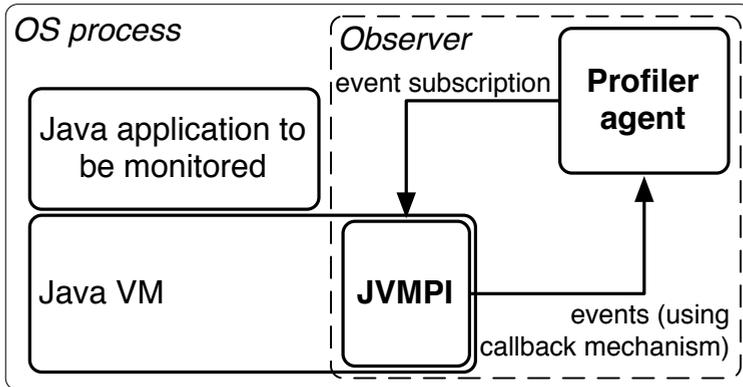


Figure 2.2: Interactions between JVMPI and the Profiler Agent

An important feature of JVMPI is its portability; its specification is independent of the virtual machine implementation. The same interface is available on each virtual machine implementation that supports the JVMPI specification. Furthermore, JVMPI does not require the virtual machine to be in debugging mode (unlike JVMDI), it is enabled by default. The Java virtual machine implementations by Sun and IBM support JVMPI.

JVMPI supports both time-driven monitoring and event-driven monitoring. This section only discusses the functionality in JVMPI that is relevant for event-driven monitoring. The profiler agent is notified of events through a callback interface. The following C++ fragment illustrates a profiler agent's event handler:

```
void NotifyEvent(JVMPI_EVENT *ev) {
    switch (ev->event_type) {
    case JVMPI_CLASS_LOAD:
        // Handle 'class load' event.
        break;
    case JVMPI_CLASS_UNLOAD:
        // Handle 'class unload' event.
        break;
    ..
    }
}
```

Listing 2.1: JVMPI event handling

The `JVMPI_EVENT` structure includes the type of the event, the environment pointer (the address of the thread the event occurred in), and event specific data:

```
typedef struct {
    jint event_type;
    JNIEnv *env_id;
    union {
        struct {
            // Event specific data for 'class load'.
        } class_load;
        ..
    } u;
} JVMPI_EVENT;
```

Listing 2.2: JVMPI event type

JVMPI uses unique identifiers to refer to threads, classes, objects, and methods. Information on these identifiers is obtained by subscribing to the defining events. For instance, the ‘thread start’ event, notifying the profiler agent of thread creation, defines the identifier of that thread and has attributes describing the thread (e.g., the name of the thread). The ‘thread end’ event undefines the identifier. For certain identifiers it is not required to be subscribed to their defining events to obtain information on the identifier. Instead, the defining events may be requested at a later time. For instance, defining events for object identifiers can be requested at any time using the `RequestEvent()` method of the JVMPI API.

JVMPI profiler agents have to be multithread aware, since JVMPI may generate events for multiple threads of control at the same time. Profiler agents can implement mutual exclusion on its internal data structures using JVMPI’s raw monitors. These monitors are similar to Java monitors, but are not attached to a Java object.

The following events are supported by JVMPI:

- *JVM start and shutdown events.* These events are triggered when the Java virtual machine starts and exits, respectively. These events can be used to initialize the profiler agent when the virtual machine is started and to release resources (e.g., close log file) when the virtual machine exits.
- *Class load and unload events.* These events are triggered when the Java virtual machine loads a class file or unloads (removes) a class. The at-

tributes of the class load event include the names and signatures of the methods it contains, the class and instance variables the class contains, etc. The class loading and unloading events are useful for building and maintaining state information in the profiler agent. For instance, when JVMPI informs the profiler agent of a method invocation it uses an internal identifier to indicate what method is being invoked. The class load event contains the information that is needed to map this identifier to the class that implements the method and the method signature.

- *Class ready for instrumentation.* This event is triggered after loading a class file. It allows the profiler agent to instrument the class. The event attributes are a byte array that contains the byte-code implementing the class, and the length of the array. Using the Java virtual machine specification, profiler agents may interpret the byte array, and change (instrument) the implementation of the class and its methods. JVMPI doesn't provide interfaces to instrument class objects though. So, all functionality needed to manipulate the array of byte-code needs to be implemented by the user of JVMPI.
- *Thread start and exit.* These events are triggered when the Java virtual machine spawns and deletes threads of control. The events attributes include the name of the thread, the name of the thread-group, and the name of the parent thread.
- *Method entry and exit.* Method entry events are triggered when a method implementation is entered. Method exit events are triggered when the method exits. The time period between these events is the wall-clock completion time of the method.
- *Compiled method load and unload.* These events are issued when just-in-time (JIT) compilation of a method occurs. Just-in-time compilation of a method compiles the (virtual machine) byte-code of the method into real (native) machine instructions. Sun's HotSpot [50] technology automatically detects often-used methods, and compiles them to native machine instructions automatically.
- *Monitor contented enter, entered, and exit.* These events can be used to monitor the contention of Java monitors (due to mutual exclusion). The monitor contented enter event is issued when a thread attempts to enter a Java monitor that is owned by another thread. The monitor

contented entered event is issued when the thread that waited for the monitor enters the monitor. The monitor contented exit event is issued when a thread leaves a monitor for which another thread is waiting.

- *Monitor wait and waited.* The monitor wait event is triggered when a thread is about to wait on an object. The monitor waited event is triggered when the thread finishes waiting on the object. These events are triggered due to waiting on condition variables for the purpose of cooperation between different threads.
- *Garbage collection start and finish.* These events are triggered before and after garbage collection by the virtual machine. These events can be used to measure the time spent on collecting garbage.
- *New arena and delete arena.* These events are sent when heap arenas (areas of memory) for objects are created and deleted. (Currently, in Java 2 SDK 1.4.2, not implemented by the JVMPI)
- *Object allocation, free, and move.* These are triggered when an object is created, released, or moved in the heap due to garbage collection.

Like the JVMDI, JVMPI also provides various utility APIs to create new system threads (which can be used in the performance tool implementation), raw monitors like (to make the performance tool thread aware), and to trigger a garbage collection cycle.

Unlike the JVMDI, JVMPI does not provide additional APIs like the JDWP and JDI APIs.

Using the event subscription API described above the JVMPI can be used to develop event-driven performance monitors. In addition to these event related capabilities, the JVMPI can also dump the heap and monitors on request. These dump capabilities can be used to develop profiler tools to find software bottlenecks, such as methods with large completion times and monitors that are often contended. Upon Java virtual machine initialization the profiler agent implementation can ask the JVMPI to create a new system thread. This thread could periodically call the `GetCallTrace()` function of the JVMPI to dump a method call trace for a given thread, or request a dump of the contents of the heap or a list of monitors.

Evaluation of the JVMDI and JVMPI

The JVMDI is meant to observe the qualitative behavior of a Java application, while the JVMPI focuses on the quantitative behavior. Both JVMDI and JVMPI can be used for studying the behavior of an application, i.e. the execution control flow (which threads are there, which methods are executed, etc.). The JVMDI can annotate this control flow information with context information such as contents of local variables, method parameters, and such. The JVMPI can annotate the control flow information with performance related events, such as the occurrence of garbage collection and locking contention.

For performance measurement JVMPI provides many useful features described above. However, there are some weak points in the JVMPI. First, a common activity in performance measurement is measuring the completion times of method invocations. The JVMPI allows the user to subscribe to method invocation events, but the user cannot give a fine-grained specification of which method invocations should be observed. So, events are generated for every method invocation in the Java virtual machine, resulting in a significant performance overhead. Secondly, the JVMPI does not provide a working API for measuring CPU times with a high-resolution. On the Linux platform the `GetCurrentThreadCpuTime()` function of the JVMPI simply returns the wall-clock time. Thirdly, while the JVMPI allows the user to intercept classes being loaded into the virtual machine, so the byte-code can be modified, the JVMPI does not provide an API to modify the byte-code; all the user gets is an array of byte-code. Fourth, JVMPI only detects and generates events for contended monitors, unlike JVMDI which allows the user to query all existing monitors. This is not an issue for performance measurement itself, but it is something to keep in mind when using JVMPI to study the performance behavior of an application. Contention for monitors may only occur for specific workloads. It is the job of the performance analyst to make sure extensive load testing (using different workloads) is done to detect monitors for which significant contention may occur.

Despite the limitations described above the JVMPI is an incredibly base for developing performance tools such as profilers and monitors. Additional functionality can be provided by the performance tool developer to work around the limitations. For instance, the tool developers can develop their own byte-code instrumentation API, use byte-code instrumentation to monitor selected method invocations, use platform dependent APIs to query performance counters to annotate the information JVMPI provides, and scan the byte-code for instructions related to monitor contention.

JVMPI does not allow us to monitor disk and network I/O and interactions with the operating system and system libraries. The developer of performance tools based on JVMPI has to implement platform specific functionality in the profiler agent, interacting with APIs outside the virtual machine, if such functionality is required.

In Chapter 3, we present our performance monitoring tool ‘JPMT’, which combines functionality from JVMPI and operating system specific APIs, working around the limitations of JVMPI and adding performance information outside the realm of the Java virtual machine such as observation of network and disk I/O and operating system thread scheduling behavior.

2.3.5 The system library layer

Sometimes instrumentation of the environment is required to obtain the required monitoring data. For instance, a library used by the application we want to collect monitoring information for can be replaced with an instrumented library. Operating systems may support a more dynamic way to instrument a library. For instance, most runtime linkers (functionality that links application code to shared libraries when the application is started) in UNIX operating systems support the LD_PRELOAD mechanism. This mechanism allows function calls to some library to be overridden by calls to a user supplied library. This user supplied library could implement wrapper functions around the real library functions the user is interested in monitoring, i.e. the user library acts as a proxy to the real library. In the wrapper functions the required instrumentation can be added.

2.3.6 The network

To monitor network socket I/O the system’s libraries could be instrumented, wrapping existing socket I/O routines as described in the previous section. A more comprehensive way of monitoring network I/O is using a ‘sniffer’.

A network sniffer hooks into the operating system’s networking layer to provide access to raw packet data of network adapters. With a sniffer we can monitor network communication between applications running on different systems. Sniffer monitoring results can be used to study network interactions between applications, measure response times of remote applications, characterize the workload an application receives via the network, and such. For example, using a network sniffer we can study the performance of a web server, focusing on its workload and overall response times.

The network sniffer does not necessarily have to run on the system where the applications are running on. It may be deployed on any machine the communication of the application is routed through.

The Packet Capture Library (PCAP) [35] is the basis of most network sniffing software on Microsoft Windows and UNIX systems. Examples of such sniffers include tcpdump and ethereal.

Many tools are available to measure response times and bandwidth of the network itself. The most well known tool to measure the response time is probably the ‘ping’ tool, available on most operating systems including Windows and UNIX, measures round-trip times on IP networks using ICMP ECHO_REQUEST and ECHO_RESPONSE packets. Examples of network bandwidth measurement tools include ‘bing’ for most UNIX systems and ‘iperf’ for UNIX and Windows.

2.4 Summary

In this chapter we introduced performance measurement activities, terminology and concepts. Furthermore, we discussed measurement difficulties and provided an overview of techniques, APIs and tools for performance measurement in applications, their supporting software layers, and hardware.

The next chapter presents our performance monitoring tool for Java applications and the Java host infrastructure middleware layer.

THE JAVA PERFORMANCE MONITORING TOOL

To build performance models of a system, a description of its execution behavior is needed. The description should include performance annotations so that the performance analyst is able to identify the behavior relevant for performance modeling. Accurate performance models require a precise description of the behavior, and good quality performance estimates or measures. [65] Our objective is to design a performance monitoring toolkit for Java that obtains both a description of the behavior of a Java application, and high-resolution performance measurements. In this chapter we present our performance monitoring tool for Java applications and the Java host infrastructure middleware layer.

This chapter is structured as follows. Section 3.1 presents our performance monitoring tool requirements. Section 3.2 presents the architecture of our performance monitoring tool. Section 3.3 explains how our tool can be used. Section 3.4 discusses implementation details of our tool. Section 3.5 discusses the intrusion of our tool. Section 3.6 summarizes this chapter.

3.1 Requirements

The Java Virtual Machine (JVM) is often used as host infrastructure middleware in current enterprise e-business applications. Parts of e-business applications are implemented in Java, including web-servers, middleware servers, and business logic. Rather than instrumenting these applications themselves, we want to monitor events that occur inside the virtual machine. This approach has several advantages; it allows so-called black-box applications (no source code availability) to be monitored and allows aspects of performance that cannot be captured by instrumenting the application itself to be captured. These include garbage collection and contention of threads for shared resources.

The monitoring tool should be able to monitor the following elements of execution behavior:

- *The invocation of methods.* The sequence of method invocations should be represented by a call-tree. To produce call-trees we need to monitor method entry and exit.
- *Object allocation and release.* In Java, objects are the entities that invoke and perform methods. The monitoring tool should be able to report information on these entities.
- *Thread creation and destruction.* Java allows multiple threads of control, in which method invocations can be processed concurrently. The monitoring tool should be able to produce call-trees for each thread.
- *Mutual exclusion and cooperation between threads.* Java uses monitors [30] to implement mutual exclusion and cooperation. The monitoring tool should be able to detect contention due to mutual exclusion (Java's synchronized primitive), and measure the duration. Furthermore, the monitoring tool should be able to measure how long an object spends waiting on a monitor, before the object is notified (`wait()`, `notify()`, and `notifyAll()` in Java).
- *Garbage collection.* Garbage collection cycles can have a significant impact on the performance of an application. Stop-the-world garbage collection, used by default in Java, introduces variability in the performance. Opening and closing network connections and bytes being transferred. Opening and closing files and reading and writing.

Further requirements:

- *Attributes to add.* The monitoring results should include attributes that can be used to calculate performance measures. For instance, to calculate the wall-clock completion time of a method invocation the timestamps of the method entry and exit are needed. The timestamps, and other attributes used, should have a high-resolution. For instance, timestamps with a granularity of 10ms are not very useful to calculate the performance of method invocations, since a lot of invocations may use less than 10ms.
- *Support modeling.* Performance modeling is a top-down process. At various performance modeling stages, performance analysts may have different performance questions. During the early modeling stages the analyst is interested in a global view of the system to be modeled. The analyst tries to identify the aspects relevant for performance modeling. In later stages the analyst has more detailed performance questions about certain aspects of the system. The monitoring toolkit should support this way of working.
- *Instrumentation: minimal overhead and automated.* Instrumentation of a Java program is required to obtain information on its execution behavior. For performance monitoring it is important to keep the overhead introduced by instrumentation minimal. So, we only want to instrument for the behavior we are interested in. During the early modeling stages, when the performance analyst wishes to obtain a global view of the behavior, the overhead introduced by instrumentation is not a major issue. However, when the analyst needs to measure the performance of a certain part of the system it is important to keep the instrumentation overhead to a minimum, since the measurements need to be accurate. This means that we need different levels of instrumentation depending on the performance questions. Manually instrumenting the Java program for each performance question is too cumbersome and time consuming. Therefore we require some sort of automated instrumentation based on a description of the behavioral aspects the performance analyst is interested in.
- *Allow development of custom tools to process monitoring results.* Tools are required to analyze and visualize the monitoring results. Since performance questions may be domain specific it's important that custom tools can be developed to process the monitoring results. Hence, the

monitoring results should be stored in an open data format. An application programming interface (API) to the monitoring data should be provided to make it easy to build custom tools.

3.2 Architecture

The architecture of JPMT is based on the event-driven monitoring approach, described in Chapter 2. JPMT represents the execution behavior of the application it monitors by event traces. Each event represents the occurrence of some activity, such as method invocation or the creation of a new thread.

The following figure, Figure 3.1, illustrates our architecture in terms of the main building blocks of our tool, and the way they are related (e.g., via input and output files).

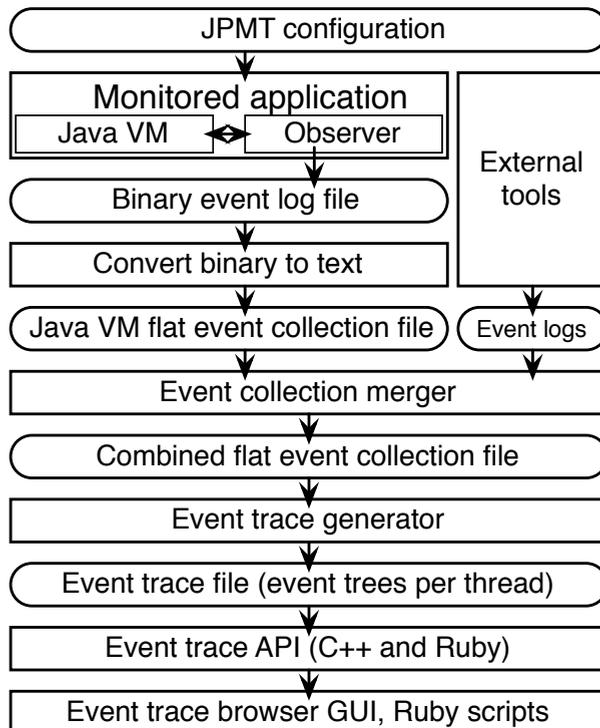


Figure 3.1: Overview of the monitoring architecture. The boxed with round edges are files. The boxes with square edges indicate software components.

First, the events of interest are specified in a configuration file. By using event filters, events can be included or excluded from the set of interesting events. For example, certain methods of a given class may be interesting, while the rest should not be monitored.

During run-time the observer component collects the events of interest. Instrumentation is added to enable generation, observation, and logging of these events. The observed events are stored in a memory-mapped binary file.

After monitoring, the binary file containing the collected events can be analyzed. The binary file is converted to a textual representation, from which event traces can be produced by the event trace generator.

The toolkit can obtain monitoring data from different sources. The primary source is the observer, which is described next section. It is also possible to use event collections from other monitoring tools to add more detail to the JPMT traces. For instance, detailed thread scheduling behavior can be obtained from operating system monitors, such as the Linux Trace Toolkit [67]. The event collections obtained from external monitoring tools can be merged with Java VM event collections.

The event traces can be accessed using an event trace API. This API is the main building block for tools that analyze or visualize event traces. The event trace API allows the performance analyst to build custom analysis tools and scripts. The API provides an object-oriented view on the event traces. Classes represent the various event types; events are instances of those classes (objects). Event attributes can be queried by invoking methods on the event objects. This API is implemented in two programming languages: C++ and Ruby. The toolkit provides two applications based on the C++ event trace API; an event trace browser and an event trace analyzer. The Ruby API allows for fast and easy development of event trace post-processing scripts.

3.3 Usage

3.3.1 Configuration

JPMT is configured using a configuration file for each application it monitors. This configuration file is read by the JPMT when the Java virtual machine that will run the Java application is started. The configuration file allows the user to choose an output file to log events to (the output option), whether or not object allocation, garbage collection, method invocation, use of Java

monitors are to be logged (that implement synchronization and cooperation mechanisms), whether or not to use byte-code instrumentation to monitor method invocations (instead of using JVMPI's method invocation monitoring mechanism), and whether or not to monitor operating system process scheduling using the Linux Trace Toolkit [67]. Using the method and thread filters the user can specify which methods and threads should be monitored, and which should not be monitored. JPMT applies these filters in the same order as they are specified in the configuration file (from top to bottom). By default all threads are monitored and no method invocations are monitored, i.e. 'include_thread *' and 'exclude_method * *' are the default thread and method filter settings. All 'yes/no' configuration directives default to 'no'. Table 3.1 summarizes the user configurable options.

Directive	Parameters	Example
output	Filename	output logs/logfile
object_allocation	"yes" or "no"	object_allocation no
garbage_collection	"yes" or "no"	garbage_collection no
method_invocation	"yes" or "no"	method_invocation yes
bytecode_rewriting	"yes" or "no"	bytecode_rewriting yes
monitor_contention	"yes" or "no"	monitor_contention no
monitor_waiting	"yes" or "no"	monitor_waiting no
lt_process_scheduling	"yes" or "no"	lt_process_scheduling no
include_thread	Thread name or wildcard	include_thread MyApplication:Pool:*
exclude_thread	Thread name or wildcard	exclude_thread *
include_method	Method name or wildcard	include_method com.test.package Hello
exclude_method	Method name or wildcard	exclude_method **

Table 3.1: JPMT configuration options

The following configuration file example logs events to log/mylogfile.bin, tells JPMT to monitor all method invocations in the com.example.test package using byte-code rewriting, and excludes all other method invocations from monitoring.

```
output log/mylogfile.bin
observe_method_invocations yes
bytecode_rewriting yes
include_method com.example.test.* *
exclude_method * *
```

Listing 3.1: JPMT configuration example

3.3.2 Processing monitoring results

This section describes the contents of the human-readable event traces. These event traces are generated from the binary monitoring log-files by

JPMT's event trace generation tool. The generated event trace file starts with some generic information, such as information on the system (processor type and speed, available memory, and timestamps of the first and last events). This generic information is followed by monitoring information of all threads of execution (if not excluded by a thread filter). For every thread basic information such as its name, parent thread, and thread group is printed, along with the timestamps when the thread started and exited (if monitoring was not stopped before thread exit). After the basic thread information, all the events that occurred in the thread are listed. Method invocations are presented in a call-tree notation (i.e. a method invocation tree). Events occurring within the context of a method invocation are listed in a sub-tree.

The events that can be monitored by JPMT are described below.

- *Thread synchronization.* This event describes the contention for Java monitors (i.e. mutual exclusion between threads). When monitor contention occurs, the following event parameters are logged: the timestamp when access to a monitor was requested, the timestamp of when the monitor was obtained, and the timestamp of when the monitor was released. In addition to these timestamps, information on the Java object representing the monitor is shown, such as the name of the class of the object, but only when object allocation monitoring has been configured in the configuration file.
- *Thread cooperation.* This event describes the occurrence of cooperation between threads in Java (i.e. condition variables that allow cooperation between threads). The event parameters include the timestamp of when the thread started waiting and the timestamp when it stopped waiting, and whether it stopped waiting because a timer expired or because it got a notification from another thread. Like the mutual exclusion event, information on the Java object representing the monitor is shown, such as the name of the class of the object, but only when object allocation monitoring has been configured in the configuration file.
- *Process (thread) scheduling information.* This provides information on the operating system process schedule changes involving the monitored thread. There are two events describing process scheduling changes in JPMT: thread-in and thread-out. Together they describe when a thread was actually running and when other processes were

running. Event parameters include the reason of the process schedule change (for instance, a change due to blocking for a resource, or a forced (involuntary) process schedule change), the timestamp when the schedule change occurred, and the threads involved. Currently, these events are not supported on other plat-forms than Linux. The Linux Trace Toolkit (LTT) is used to obtain this monitoring information.

- *Method invocations.* A method invocation event includes the timestamp of when it occurred, the used wall-clock time in microseconds, the used CPU time in microseconds, and whether or not the method was compiled from byte-code into native code by the Java virtual machine. Method invocations are represented in call-tree notation. The wall-clock and CPU time event parameters of the caller method include the wall-clock and CPU time of callee methods. Mutual exclusion, cooperation, garbage collection, process scheduling events, and such, are shown within the method in which they occurred.
- *Object allocation and release.* The object allocation and release events describe the allocation and release of dynamically allocated memory for objects. Along with the timestamp of when the event occurred, information on the object involved, such as the name of the object class, is also shown.
- *Garbage collection.* This event describes the occurrence of garbage collection. Parameters include the start and end timestamps of the garbage collection cycle.

3.3.3 Presentation of monitoring results

The observer produces a binary log file containing the collected events. From this event collection, event traces can be generated. The toolkit provides an event trace generator that produces an event trace for each program thread. Tools are needed to visualize and analyze these event traces.

JPMT provides an event trace API that can be used to implement event trace processing tools. The API provides an object-oriented view on the event traces. C++ classes represent the various event types; events are instances of those classes (objects). Event attributes can be queried by invoking methods on the event objects.

We have implemented two tools on top of this API: (i) an event trace browser (or visualizer), and (ii) a command-line tool for analyzing event traces.

The event trace browser provides a simple graphical user interface for browsing the event trace. Events, including performance attributes, are displayed on the screen in a tree-like structure. The user is able to expand and collapse sub traces, depending on the level of detail the user requires.

Figure 3.2 presents a screenshot of the event-trace browser. It is a trace fragment of a request dispatching thread of a CORBA implementation. The first column contains the name of the methods that are being invoked, including caller-callee relationships (tree hierarchy). The second column contains the measured wall-clock completion time in microseconds. The third column contains the measured CPU usage in microseconds. The fourth column shows queuing information of the request; before being dispatched requests sit in a queue waiting for a dispatcher thread to become available. The column shows that request number 87 spent 749066 microseconds in the request queue and that there are still 4 requests in the queue (excluding request number 87). The queuing information is obtained by replaying actions in the event trace related to the request queue, i.e., the en-queue and de-queue operations of the queue.

The screenshot shows a window titled "A simple event trace viewer: ORBacus:ThreadPool-0:Dispatcher-2 (env ptr is 0x4ecd8d04) P3-700MHz". The main area displays a tree view of events. The columns are "Event", "Wall usec", "CPU usec", and "Queuing". The tree structure shows a sequence of method calls, with some expanded to show sub-calls. The "Queuing" column contains information like "no.87 #4 749066us".

Event	Wall usec	CPU usec	Queuing
com.ooc.OB.ThreadPool#get()	17	16	
com.ooc.OB.DispatchRequest_Impl#invoke()	590085	682	no.87 #4 749066us
com.ooc.OB.PortableServer.POA_Impl#_OB_dispatch()	590067	664	
com.ooc.OB.PortableServer.ActiveObjectOnlyStrategy#locate()	30	28	
com.ooc.OB.PortableServer.POA_Impl#_OB_preinvoke()	61	59	
com.ooc.OB.PortableServer.ServantDispatcher#dispatch()	589943	540	
net.qahwah.jpmt.examples.ORBacus.PerformanceTest1POA#_invoke()	589748	346	
net.qahwah.jpmt.examples.ORBacus.PerformanceTest1POA#_OB_op_ping_with_wait()	589713	310	
net.qahwah.jpmt.examples.ORBacus.PerformanceTest1_Impl#ping_with_wait()	589472	68	
com.ooc.OB.PortableServer.ServantDispatcher#createReply()	200	198	
com.ooc.OB.Upcall#postMarshal()	154	152	
com.ooc.OB.ThreadPool#get()	18	16	no.90 #6 799373us
com.ooc.OB.DispatchRequest_Impl#invoke()	589944	634	

Figure 3.2: Event-trace browser screenshot

The command-line tool can be used in scripts that post-process the event traces for specific experiments, for example, to obtain input data for GNUplot (a tool to generate plots). The command-line tool can also be used to export the event traces to a human readable text format.

We plan to add language bindings for the Java and Ruby programming languages to the API, such that performance analysis applications can be written in those languages too.

3.4 Implementation

3.4.1 The JPMT Virtual Machine Observer

Our first profiler agent prototype logged events in a human readable (text) ASCII data format. Threads writing to the log file needed to obtain a global lock for the file. This lock is held while writing the event to the file.

Our second prototype replaced the text format with a binary format, which is significantly faster. The binary log file is memory-mapped into the profiler agent's address space. Different threads can write simultaneously to the memory map. Of course, threads should not log events in the same location of the memory map. A position in the memory map is assigned using a position counter. Incrementing the position counter in the memory map requires mutual exclusion. The position counter is increased with the size of the data that is to be written.

The combined use of a binary data format, memory-mapped I/O, and mutual exclusion to a position counter instead of the whole file, makes this solution much faster than using text files. There is a downside though; portability is sacrificed. Each operating system may implement different interfaces for memory-mapped files. For example, on POSIX `mmap(2)` [57] is used, while Microsoft Windows has `CreateFileMapping(Win32)`.

Initialization, configuration, and maintenance of internal tables

During its initialization the profiler agent reads a user specified configuration file, containing event filters. For example, the user may indicate which classes and methods are to be monitored, or whether locking contention is to be monitored. The configuration file format was described earlier in Section 3.3.1.

After specifying the log file, the above configuration turns on observation of monitor contention, object allocation, and method invocations. Furthermore, it indicates that byte-code rewriting is to be used to instrument for method invocation monitoring. Finally, it specifies the methods to be observed, using include and exclude filters. These filters take two arguments: the class name and the method name. Wildcards are allowed in these filters.

When the virtual machine loads a class file the JVMPI informs the profiler agent about it using a 'class load' event. This event has the following attributes:

```
struct {
```

```
char *class_name;
char *source_name;
jint num_interfaces;
jint num_methods;
JVMPI_Method *methods;
jint num_static_fields;
JVMPI_Field *statics;
jint num_instance_fields;
JVMPI_Field *instances;
ObjectID class_id;
} class_load;
```

Listing 3.2: JVMPI class_load structure

The profiler agent uses the attributes from the ‘class load’ event to build hash tables that map class and method identifiers to class and method information, respectively. The hash tables only keep information for classes and methods that are to be monitored. A ‘class unload’ event causes the information related to the class to be removed from the hash tables. The following code fragment depicts how the information on classes and methods are stored in hash tables.

```
class ClassInfo
{
public:
    jobjectID classID;
    char *name;
    bool dumped;
};

class MethodInfo
{
public:
    jmethodID methodID;
    ClassInfo *classInfo;
    char *name;
    char *signature;
    bool dumped;
};

typedef hash_map<
    jobjectID ,
    ClassInfo *,
    hash<jobjectID> > ClassMap;
```

```
typedef hash_map<
    jmethodID,
    MethodInfo *,
    hash<jmethodID> > MethodMap;
```

Listing 3.3: JPMT class en method information

Class and method information is written to the log file on demand. For instance, when a monitored method is invoked for the first time, information on that method is logged. The ‘dumped’ fields in the ClassInfo and MethodInfo classes indicate whether the information has been logged or not. The following class information is logged:

- The ‘classID’ field, which uniquely identifies a class.
- The ‘name’ field, which contains the name of the class.

The following information is logged for methods:

- The ‘methodID’ field, which uniquely identifies the method.
- The class identifier of the class the method is part of, obtained via the ‘classInfo’ reference to the object that holds the class information.
- The name of the method, which is stored in the ‘name’ field.
- The signature (return type and types of the parameters) of the method, which is stored in the ‘signature’ field.

Monitoring thread spawning and deletion

The profiler agent records the spawning and deletion of every thread of control, using the ‘thread start’ and ‘thread exit’ JVMPI events. A thread is identified by its environment pointer (thread_env_id). The profiler agent maintains a hash mapping of this environment pointer to a structure that contains information on the thread. When the thread exits the profiler agent removes the thread information from the hash map. The following code fragment shows the attributes of the ‘thread start’ event, and how thread information is stored in a hash table.

```
struct {
    char *thread_name;
    char *group_name;
    char *parent_name;
```

```
    jobjectID thread_id;
    JNIEnv *thread_env_id;
} thread_start;

class ThreadInfo
{
public:
    char *name;
    char *groupName;
    char *parentName;
};

typedef hash_map<
    JNIEnv *,
    ThreadInfo *,
    hash<JNIEnv *> > ThreadMap;
```

Listing 3.4: JPMT thread information

Spawning and deletion of threads is recorded in the event log file. The following information is logged when a thread is spawned:

- The environment pointer of the thread.
- The object identifier of the thread, when the profiler agent is configured to monitor object allocations. The object identifier can be used to obtain type information of the thread object, e.g., the name of the class of the thread object.
- The name of the thread, the name of the parent thread, and the name of the thread group this thread is part of.
- A timestamp, representing the time the event occurred.

When a thread is deleted the following information is logged:

- The environment pointer of the thread.
- A timestamp, representing the time the event occurred.

Monitoring method invocations using JVMPI method entry and exit events

JVMPI notifies the profiler agent of method invocations using the method entry and method exit events. Both JVMPI events have one event specific attribute, the method identifier.

```
struct {  
    jmethodID method_id;  
} method;
```

Listing 3.5: JPMT method information

When the profiler agent is notified of a method entry it executes the following algorithm:

1. Obtain the thread local storage to store per-thread data. The profiler agent creates this thread local storage when it processes the first method entry in the thread.
2. If the method hash map doesn't contain information for the method identifier (meaning that the method is not to be monitored) then a counter in the thread local storage is increased and the event handler exits. This counter represents the number of un-logged method invocations in the thread of control. It can be used to correct the event trace for perturbation. For example, if a logged method calls 5 other methods that are filtered out (not logged), the CPU usage and completion time of that method includes the time for observing and filtering the un-logged method invocations. By keeping track of the number of un-logged invocations, we can subtract the costs of observing and filtering out these method invocations from the CPU usage and completion time of the logged method.
3. If the method hash map does contain information on the method identifier then information on the method and its class is logged if it hasn't been logged before. The un-logged method-invocation counter is reset and its old value is pushed on a stack in the thread local storage. Subsequently, the information on the method entry is logged, including timing related attributes, and the event handler exits.

A logged method entry contains the following attributes:

- The environment pointer of the thread that executes the method invocation.

- The identifier of the method.
- The value of the un-logged method invocations counter, before it has been reset.
- Information needed to determine the CPU usage of the method.
- A timestamp, representing the time the event occurred.

Unfortunately, each operating system implements interfaces to obtain CPU timing information differently. For example, POSIX (a UNIX standard) offers the `times(2)` call, Windows offers `GetThreadTimes(Win32)` and `GetCurrentThreadCPUTime(Win32)`, BSD UNICES and derivatives have `getrusage(2)`, and Sun Solaris has `gethrtime(2)` and `gethrvtime(2)`.

JVMPI offers `GetCurrentThreadCpuTime()`, which is supposed to return the CPU time in nano-seconds. However, on Linux it returns the same value as `gettimeofday(2)` does: the current wall-clock time in micro-seconds.

Often, CPU time information has a 10ms resolution, e.g. POSIX' `times(2)`, BSD's `getrusage(2)`, and Windows' `GetThreadTimes(Win32)` have a 10ms granularity. This is too coarse to be used as a performance measure for Java method invocations. This is caused by the frequency of the clock interrupt timer. Most operating systems are configured to generate 100 clock interrupts per second.

There is a solution for this problem: architecture specific hardware performance counters. All modern micro processors, including Intel's Pentium family, IBM/Motorola's PowerPC, and Compaq/DEC's Alpha, implement such performance counters.

These hardware counters don't have the granularity problem, but they still have an interfacing problem: there is no common interface to access these counters. Libraries such as PAPI [7] provide a common interface to these counters. The current implementation of the profiler agent doesn't use such a library, but implements similar functionality.

The profiler agent accesses the hardware performance counters using an operating system specific device driver. On the Linux operating system Mikael Pettersson's PerfCtr [44] package is used.

Besides propagating hardware performance counter values to user-land, these device drivers could also implement virtual performance counters for each process (thread). Virtual counters are only incremented when a process is executing, not when it is waiting in the operating system's process

scheduler queue. So, in contrast to global counters, these counters provide precise timing information for the process.

On the Intel platform information that can be obtained using hardware performance counters includes the number of processor cycles since the processor was booted [11]. We use this counter to calculate the CPU usage of a method invocation. The information that can be obtained using hardware performance counters can be much more detailed, e.g. efficiency of the caches, and efficiency of branch prediction. For our purposes this is far too detailed.

The CPU usage of a method can be calculated by subtracting the number of processor cycles at method entry from the number of processor cycles at method exit.

Monitoring contention during mutual exclusion

The JVMPI ‘monitor contented enter’, ‘entered’, and ‘exit’ events are logged with timestamps, the environment pointer of the thread, and the object-id of the Java object that is associated with the monitor.

Monitoring cooperation

The JVMPI ‘monitor wait’ and ‘waited’ events are logged with timestamps, the environment pointer of the thread, and the object-id of the Java object that is associated with the monitor.

Object allocation

In case information on Java objects is needed, for instance when there is contention for a monitor, the observer queries JVMPI for object information. This information is stored in the observer’s internal data structures. To keep this information valid the observer also needs to monitor ‘object free’ events, and ‘object move’ events (an object may be moved to another address during garbage collection). The object information is also logged, so that the analyzer and visualizer tools can display object information of objects associated to monitor contention and cooperation events.

Java byte-code instrumentation

Processing method entry and exit notifications sent by the JVMPI for every method invocation introduces a lot of overhead. At minimum, two hash table

lookups are required (to see whether or not to log the method entry and exit). To reduce the overhead introduced due to monitoring method invocations we have also implemented another monitoring approach: instrumentation of the byte-code of method implementations. This instrumentation mechanism only inserts instrumentation in methods that we want to monitor. This is different from JVMPI's method entry and exit events, which are triggered for every method invocation, similar to the interceptor design pattern [13].

JVMPI provides the 'class ready for instrumentation' event, which can be used to insert instrumentation in Java classes. However, no user-friendly interface is provided to change the implementation of the classes. For Java, libraries are available that provide the user with APIs for rewriting the implementation of classes, such as the Byte Code Engineering Library (BCEL) [38]. Unfortunately, we cannot use these libraries since the profiler agent has to be implemented in C or C++. Rewriting classes before run time is not an option, since that solution is in conflict with the requirement that the insertion of instrumentation should be transparent to the user, and be done at run-time.

We choose to implement the byte-code instrumentation ourselves, in the profiler agent. The 'class ready for instrumentation' event provides us with a byte array that contains the compiled class object. The format of the class object is described in the Java Virtual Machine Specification [37]. The following pseudo-code fragment describes the structure of class objects. Data types represented are: u2 and u4, which are 2-byte and 4-byte unsigned numbers; cp_info, field_info, method_info, and attribute_info, which are structures that describe constant pool entries (e.g., names of methods), variables, methods, and attributes, respectively.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count - 1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
}
```

```
u2 attributes_count;  
attribute_info attributes[attributes_count];  
}
```

Listing 3.6: Java ClassFile

The byte-code rewriter takes the original byte-code array, rewrites it, and returns the rewritten byte-code array to the virtual machine. The rewriting algorithm consists of the following steps (the rewriting algorithm is only executed if the class contains methods that we should monitor):

- All bytes of the class are copied to a newly allocated byte array that keeps the instrumented class.
- The constant pool of the class is parsed and several administration tables are built, containing information about constants. The constant pool contains, for instance, references and names of classes the class refers to, and references and names of methods it calls.
- New constant pool entries for the class and methods that implement the instrumentation (i.e. the call-back handlers in the observer that will be called for each instrumented method entry and exit) are added to the instrumented class.
- Subsequently, the list of methods is iterated. If a method is to be monitored, instrumentation is inserted at the method entry point and before every method exit. The instrumentation contains byte-code that notifies the profiler agent of method entry and exit. The insertion of byte-code instructions renders branching offsets, and position counters of exception handlers invalid. The rewriting algorithm fixes these relative and absolute addresses. In addition, the maximum stack size may need to be updated, since the instrumentation contains method invocations of the profiler agent's method entry and exit handlers. Method invocations use the stack to store parameters.

Below we illustrate the rewriting algorithm by showing how a simple Java method is rewritten. First, the Java source code:

```
public static int test2b(int i) {  
    if (i == 2) return test2a(i+1);  
    else return test2a(i);  
}
```

Listing 3.7: Java method example

Now we show the byte-code the Java compiler produced:

```

PC   OPCODE          OPERANDS
0x0  0x1a  iload_0
0x1  0x05  iconst_2
0x2  0xa0  if_icmpne    0x0 0xa
0x5  0x1a  iload_0
0x6  0x04  iconst_1
0x7  0x60  iadd
0x8  0xb8  invokestatic 0x0 0x2
0xb  0xac  ireturn
0xc  0x1a  iload_0
0xd  0xb8  invokestatic 0x0 0x2
0x10 0xac  ireturn

```

Listing 3.8: Java byte-code before instrumentation

If the performance analyst indicated in JPMT's configuration file that the method `test2b()` is to be monitored, JPMT's byte-code rewriting engine inserts instrumentation at the method entry point and every method exit point. In this example there are two exit points, because of the two return statements in the if statement. The instrumentation inserted by JPMT is depicted by a '*' in the code fragment below. The instrumentation consists of two byte-code instructions. First, we push the method identifier JPMT allocated for this method (in this case the operand `0x01`) onto the stack using the `bipush` instruction. The `sipush` instruction is used instead of `bipush` once we start monitoring more than 256 methods (`bipush` pushes a byte, `sipush` a short integer). Then we invoke, using the `invokestatic` method, a call-back handler in the JPMT observer for either a method entry (operands `0x00 0x1E`) or method exit (operands `0x00 0x1F`). Recall from the instrumentation algorithm explanation above, that the algorithm adds class and method information for JPMT's call-back handlers to the instrumented class. The operand values of the `invokestatic` byte-code instruction are index values of the instrumentation call-backs in the instrumented class' constant pool.

```

PC   OPCODE          OPERANDS
0x0  0x10  bipush      0x01      *
0x2  0xb8  invokestatic 0x0 0x1e  *
0x5  0x1a  iload_0
0x6  0x05  iconst_2
0x7  0xa0  if_icmpne    0x0 0xf
0xa  0x1a  iload_0
0xb  0x04  iconst_1
0xc  0x60  iadd

```

```

0xd  0xb8  invokestatic 0x0 0x2
0x10 0x10  bipush      0x01      *
0x12 0xb8  invokestatic 0x0 0x1f  *
0x15 0xac  ireturn
0x16 0x1a  iload_0
0x17 0xb8  invokestatic 0x0 0x2
0x1a 0x10  bipush      0x01      *
0x1c 0xb8  invokestatic 0x0 0x1f  *
0x1f 0xac  ireturn

```

Listing 3.9: Instrumented Java byte-code

3.5 Intrusion

The overhead of monitoring depends on the amount of instrumentation. We identified two distinct levels of monitoring: (i) exploring the behavior of the monitored software and (ii) measuring performance of parts of the software to answer specific performance questions. For exploring behavior it doesn't really matter how much overhead is introduced. For performance measurement however, we like to minimize the overhead, since it disrupts the measurements. We can do this by filtering out events we're not interested in.

Monitoring of method invocations causes the most overhead. On a Pentium III 700 MHz system the overhead for monitoring and logging a method invocation is 6 microseconds (on an AMD Athlon XP 2000 at 1.67 Ghz we measured 5 microseconds) when the byte-code rewriting algorithm is used to instrument methods. This can be split evenly in 3 microseconds for monitoring the method entry event, and 3 microseconds for monitoring the method exit event. Logging itself costs 1 microsecond for each event (excluding disk flushes of the memory map). The other 2 microseconds per event are caused by the invocation of our method invocation event handler and obtaining timestamps and CPU performance counters. To compare, a Java method that does a print-line on the screen (`System.out.println()`), costs 16us (on the AMD machine we measured 17 microseconds). We provide a small utility to measure overhead caused by the instrumentation on other platforms.

3.6 Summary

The construction of performance models requires insight in the execution behavior (how does the application work performance-wise) and good-quality

performance measurements (how fast does the application perform its work, and which resources are used) [65]. The insight on the execution behavior can be used to develop the structure (topology) of the performance model. The performance measurements can be used to identify the parts of the execution behavior that need modeling, to validate performance models, and they can be used as performance model parameters.

Our Java Performance Monitoring Tool (JPMT) is based on event-driven monitoring [32]. JPMT represents internal execution behavior of Java applications (also middleware and web servers) by event traces, where each event represents the occurrence of some activity, such as thread creation, method invocation, and locking contention. JPMT supports event filtering during and after application execution. Each event is annotated by high-resolution performance attributes, e.g., duration of locking contention, and CPU time usage by method invocations. The instrumentation required for monitoring the application is added transparently to the user during runtime. Source code of the application is not required, making the tool suitable for monitoring so-called black-box applications for which no source code is available. Overhead is minimized by only adding instrumentation for events the user (of the tool) is interested in and by careful implementation of the instrumentation itself.

In the next chapter we discuss the inner workings of CORBA object middleware and thread scheduling, and present our performance models for CORBA object middleware.

PERFORMANCE MODELING OF CORBA OBJECT MIDDLEWARE

In this chapter we discuss the inner workings of CORBA object middleware and thread scheduling, and present our performance models for CORBA object middleware.

This chapter is structured as follows. Section 4.1 introduces CORBA middleware. Section 4.2 discusses the specification and implementation of threading in CORBA. Section 4.3 presents our performance models. Section 4.4 discusses how we generate workload in our experiments. Section 4.5 compares the throughput of the various threading strategies. Section 4.6 discusses the impact of marshaling in CORBA. Section 4.7 discusses performance modeling of the thread scheduler of the operating system. Section 4.8 summarizes this chapter.

4.1 CORBA object middleware

The Common Object Request Broker Architecture (CORBA) [41] specified by the Object Management Group (OMG) is the de-facto object-middleware standard. CORBA mediates between application objects that may live on different machines, by implementing an object-oriented RPC mechanism. This allows application objects to communicate with remote objects in the

same way as they communicate with local objects. Besides this *object location transparency*, CORBA also implements *programming language transparency*. Object interfaces are specified in an interface description language (IDL). These IDL interfaces are compiled to so-called *stubs* and *skeletons*, which act as proxies for the client and server objects, respectively. The client and server objects may be implemented in different programming languages, for instance Java and C++.

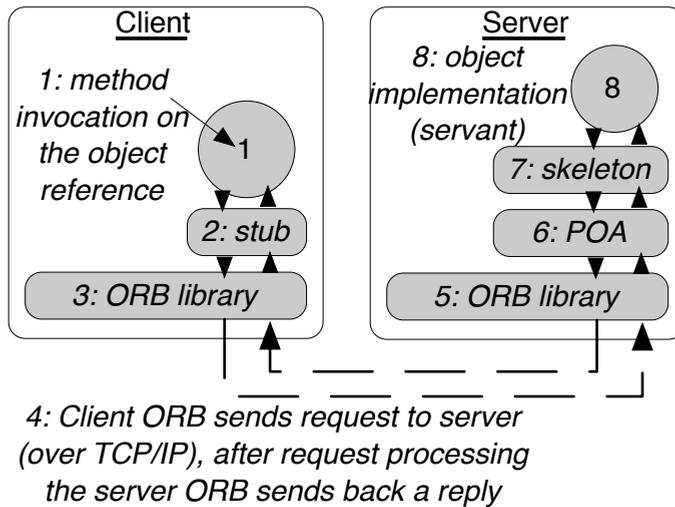


Figure 4.1: Anatomy of a CORBA method invocation

To invoke a method on a remote CORBA object, the following sequence of steps is taken, as illustrated by Figure 4.1 (illustrating a two-way request-and-reply method invocation).

1. *Remote method invocation.* The client obtains the *object reference* of the remote target object and performs a method invocation on it as if the object were a local (e.g. Java) object.
2. *Stub processing and marshaling.* What really happens is that the client invokes the method on the stub, which is the local proxy of the remote target object. A reference to the proper stub is obtained from the object reference of the target object. The stub constructs a CORBA request object and translates the method invocation parameters, which are

- expressed using programming language, operating system, and architecture specific data types, to a *common data representation* (CDR). This translation process is called marshaling. The marshaled data is added to the request object. Subsequently, the request object is forwarded to the client-side object request broker (ORB) library.
3. *Client-side ORB processing.* The client-side ORB library uses a TCP/IP connection to communicate with the server-side ORB library. The address of the server-side ORB is obtained from the target object's object reference. The object reference was created by a *portable object adapter* (POA) in the server-side ORB. Each object is managed by exactly one POA. A POA implements the adapter design pattern [13] to adapt the programming language specific object interfaces to CORBA interfaces, making the target object implementation accessible from the ORB. The POA manages references to active objects in the active object map. This map associates object identifiers with object implementations. Object implementations are called *servants*. The object reference contains server information, such as hostname and port number, the name of the POA, and the object identifier of the target object.
 4. *ORB communication.* The client-side ORB sends the request object to the server-side ORB.
 5. *Server-side ORB processing.* The server-ORB obtains the target object's POA and object identifiers from the request object, and forwards the request to the POA managing the target object.
 6. *Object adaptor processing.* The POA looks up the servant in its active object map using the object identifier and forwards the request to the skeleton of the target object.
 7. *Un-marshaling.* The skeleton un-marshals the method invocation parameters and looks up the method implementation.
 8. *Method invocation in the object implementation.* The skeleton invokes the request on the proper method implementation inside the object implementation of the target object.
 9. The road back, creating a reply and sending it to the client. When the method invocation returns, the skeleton creates a CORBA reply object, marshals the return parameters, and inserts the return parameters in the reply object. The reply object is forwarded to the server-side ORB.

Subsequently, the server-side ORB forwards the reply object to the client-side ORB. Then, the client-side ORB forwards the reply object to the stub, and finally, the stub un-marshals the return parameters and forwards those to the client.

This description is a bit simplified. A full description of what happens during a method invocation is available from [41] and from many books on the topic, such as [29].

4.2 Specification and implementation of CORBA threading

In this section we describe what threading strategies are, and what kind of threading strategies are implemented in popular CORBA implementations. A CORBA threading strategy determines how communication and execution of requests take place. A thread is a light-weighted process, having its own execution context (processor and memory state), but sharing memory and file descriptors with other threads inside the process. By using multiple threads, an ORB can receive and execute multiple requests concurrently. Threading allows for more efficient use of the system's resources, to achieve a higher request throughput.

4.2.1 Threading in the OMG CORBA specification

The CORBA specification [41] specifies multi-threading policies for portable object adapters (POA). It doesn't specify how ORBs should implement threading and what threading strategies they should implement. Multi-threading implementation in the ORB itself is left as platform-specific. The POA threading policies are there to 'support threads in a portable manner across those ORBs' that implement multi-threading. The CORBA specification doesn't require the ORB to be multi-threaded, however ORBs are required to implement the POA threading policies if they are multi-threaded.

The specification defines three threading policies for portable object adapters: the Single Thread Model, the ORB Controlled Model, and the Main Thread Model. These threading policies determine what concurrency constraints the POA will enforce while executing requests. The *ORB Controlled Model* (the default) doesn't impose any concurrency constraints, requests to objects managed by the POA may execute in parallel. The *Single Thread Model* only allows one request to objects managed by the POA at a time (the POA

dispatches requests sequentially). Requests to a single threaded POA are mutually exclusive, a lock (mutex) is introduced (by the ORB vendor in the middleware implementation), and locking contention may occur. The *Main Thread Model* only allows one request at a time to be processed by any POA with the main thread policy. Requests to any POA with main thread policy are mutually exclusive, a lock (mutex) is introduced, and locking contention may occur. The lock is shared by all main threaded POAs. The Main Thread Model was introduced in the CORBA 2.4 specification. In this work we focus on the ORB Controlled Model (the default).

4.2.2 Server side thread categories

On the server side of a CORBA application we can distinguish the following kinds of threads: acceptor threads, receiver threads, dispatcher threads, application specific threads, and administrative threads.

A common way of implementing I/O, i.e. connection setup and exchange of GIOP (General Inter-ORB Protocol, CORBA's messaging protocol) messages between client and server, is to use a specific acceptor thread for listening to incoming connections to CORBA objects (really the POA manager that manages the object). Typically, for every new connection a receiver thread is spawned that will handle the exchange of GIOP messages.

GIOP messages are received in receiver threads. In some threading strategies the method invocations to CORBA objects are executed in receiver thread. During this time the receiver thread cannot handle new GIOP messages, such as a new method invocation request. For applications that use method invocations with longer completion times it may be better to 'delegate' the execution to a separate thread: a dispatcher thread. The dispatcher thread will then handle the dispatching of the method invocation request to the servant. The method invocation will be executed in the context of the dispatcher thread. After delegating the method invocation request to the dispatcher thread, the receiver thread is free to receive the following GIOP message. Some time may elapse between the receiver thread handing over the request to a dispatcher thread, and the dispatcher thread actually handling the request.

The application running on top of the ORB can be multi-threaded also. Usually the 'ORB Controlled Model' threading policy is used in POAs that manage servants for that application. The application (including the servants belonging to that application) is designed to handle requests concurrently – the application takes care of implementing critical sections (mutual exclusion)

where required.

Besides the above mentioned kinds of CORBA related threads several Java virtual machine related administrative threads, such as Java garbage collection threads, will be active.

4.2.3 Threading strategies in ORBacus

We have used the ORBacus CORBA middleware by IONA to conduct our performance experiments. The reason for using ORBacus is threefold. First, ORBacus provides a complete set of threading strategies, which is most important to make a performance comparison of the most widely used threading strategies. Second, ORBacus is widely applied and available for both C++ and for Java. Third, ORBacus is both commercially and academically available. Other implementations, such as Orbix, Visibroker, JacORB and OpenORB, support comparable threading strategies with comparable features. We will describe the differences between the different CORBA implementations where applicable.

The *thread-per-client* threading strategy uses receiver threads for both receiving and dispatching the requests to servants. Because requests are dispatched by the same thread as they are received in, following requests by the client are blocked until the receiver thread is done processing the current request. Effectively, only one request per client is active in user code, i.e. in the servants. Each client-server connection has its own receiver thread, so multiple requests from different clients can be active in user code.

The *threaded threading* strategy is the same as the thread-per-client model, but with an additional constraint: only one request may be active in user code. The ORB serializes requests from the receiver thread when dispatching them to servants, using a global ORB mutex. While in the thread-per-client model one request per client could be active in user code (so multiple requests could be active if multiple clients are connected to the ORB), only one request ORB-wide can be active in user code in the threaded model.

Both the thread-per-client and the threaded threading strategies use the receiver thread for receiving and dispatching requests to servants. The *thread-per-request* threading strategy separates receiving requests and dispatching requests in separate threads. After receiving a request from a client, the receiver thread creates a new dispatcher thread. That thread dispatches the request to the servant, and sends the reply back to the client. Meanwhile the receiver thread can receive new requests (and dispatch them in new

dispatcher threads). Thus, in the thread-per-request model multiple requests from the same client can be active in user code.

The *thread-pool* threading strategy is a refinement of the thread-per-request model. It addresses several issues that may arise when using the thread-per-request model. First, the number of dispatcher threads is not bounded in the thread-per-request model. This can lead to an uncontrolled growth of the number of dispatcher threads. With many dispatcher threads active, the context switch overhead becomes very large, and even trashing behavior may occur, where the machine is mostly busy switching contexts rather than executing user code. Another problem of the thread-per-request model is that thread creation and destruction is needed for each request. Especially for requests that don't require a large amount of processing time, the added overhead of thread creation and destruction is relatively large, thus leading to inefficiencies. The thread-pool model addresses these issues by pre-allocating a fixed number of dispatcher threads when the ORB starts. Instead of creating new dispatcher threads, the receiver thread en-queues requests in the FIFO request queue of the dispatcher thread pool. The dispatcher thread pool assigns a dispatcher thread to process the request when an idle (non-working) dispatcher thread is available.

4.2.4 Threading strategies in other CORBA implementations

All ORBs discussed in this section implement the thread-pool model, albeit in different variations. ORBacus implements a global thread pool (shared between all POAs) with a fixed size. The thread pool is allocated (all threads are created) when the ORB starts. Idle threads are not killed, but remain in the thread pool. OpenORB and Orbix also implement a global thread pool, but with a minimum and maximum number of threads. When the ORB starts, a minimum number of threads are allocated. The actual number of threads in the thread pool will vary between the minimum and maximum depending on the load. Idle threads are killed when there are no requests available for processing and when the thread pool is larger than the minimum size. JacORB implements thread pools per POA. These thread pools have a maximum size and a maximum number of idle threads. Threads are killed when they return and find that the maximum number of idle threads has been reached. Visibroker implements thread pools per POA. These thread pools have a minimum and maximum size. Visibroker implements the leader/follower design pattern [47] to dispatch requests to servants. The request is dispatched to the servant in the same thread as it was received in; a thread is taken from the thread pool to listen for new requests. So, contrary to other

thread pool implementations described in this section, the request will not be handed over to a dispatcher thread (taken from the thread pool). Sun's CORBA implementation built-in JDK 1.4.1 also implements a thread pool, but this pool has an unbounded size. Idle threads have a two-minute timeout before they're killed. The thread pool is global (for all POAs). All non-fixed thread pools grow towards the maximum number of threads when a request arrives and no idle thread is available.

ORBacus, Orbix and Visibroker are the most flexible CORBA implementations for Java when it comes to threading models. All three implementations offer various threading models. Orbix offers a 'ThreadFilter' API to the CORBA application programmer, where custom threading models can be implemented. Visibroker offers a similar facility to plug-in custom threading models. Visibroker and ORBacus seem to be the only ORBs implementing threading models that dispatch requests in the same thread as they are received in (thread-per-session in Visibroker, thread-per-client and threaded in ORBacus).

Implementation	Dispatching models	Thread-pool	Supported threading strategies
JacORB 2 beta	Separate request processor threads.	Thread-pool for each POA, with a maximum number of idle threads and a maximum size.	Thread-pool.
OpenORB 1.4.0	Separate request processor threads.	An ORB-wide thread pool, with a minimum and maximum size.	Thread-pool.
ORBacus 1.4.1	Supports both the same thread model and separate request processor thread model.	An ORB-wide thread pool, with a fixed size.	Thread-per-client, threaded, thread-per-request & thread-pool.
Sun JDK ORB 1.4.1	Separate request processor threads.	An ORB-wide thread pool, with an unbounded size and a 2-minute timeout before idle threads are killed.	Thread-pool (basically thread-per-request with temporary caching of non-idle threads). No support for the single threaded POA policy.
ORBix 3.3	Separate request processor threads.	An ORB-wide thread pool, with a minimum and maximum size.	Thread-per-process (thread-per-request), thread-per-object (thread-per-servant), thread-pool & a thread-filter API allowing custom threading models to be implemented.
VisiBroker 4.5	Supports the same thread model for the thread-per-session strategy, the leader-follower model for the thread pool strategy. The threading model is configurable per POA.	Thread-pool for each POA, with a minimum and maximum size and a timeout for idle threads.	Thread-per-session (thread-per-client) & thread-pool (but uses leader/follower design pattern).

Table 4.1: Threading strategies of popular CORBA implementations

Table 4.1 contains one thread model that has not been discussed above: the thread-per-object threading model implemented by Orbix. In the thread-per-object model each servant has its own thread for executing requests. The ORB thread that received the request en-queues the request in the FIFO request queue of the target servant. The servant thread processes requests from that queue one by one. The thread-per-object threading model thus restricts the number of active requests in a certain object to one at a time.

All Java ORBs discussed in this section implement the same I/O threading model: a listener (or acceptor) thread listens for new connections and receiver (or receptor) threads are created for each connection.

4.3 Performance models of threading strategies

To highlight the differences in the dynamic behavior of the four threading strategies we present performance models for each of the threading strategies. We present the performance models in an extended queuing network notation [36] [32]. Figure 4.2 depicts some modeling constructs that require further explanation. First, a thread is modeled by a service center inside a square. All steps inside the dashed box are executed in the context of that thread. These steps contribute to the holding time of the thread. While the thread is busy, requests can queue in the FIFO queue in front of the thread. A thread pool is a group of threads that can execute some steps depicted in the dashed box. A thread pool is modeled as a multi-server with a FIFO queue in front of it. A mutex is modeled by a queue and a dashed box denoting the processing steps inside the critical section of the mutex. Finally, we use a kind of zooming construct to denote the steps taking place inside a CORBA portable object adapter. We separate the logical and physical resource layers. The logical resource layer contains the threads and mutexes, while the physical resource layer contains the CPUs. The CPU service demands described in the logical resource layer are executed on the CPUs in the physical resource layer.

Our performance models focus on CORBA middleware implementations and their threading behavior, and do not include performance aspects of the network, disk I/O and memory resources in the physical resource layer. This does not mean we think these are not important performance aspects. For instance, network latency is of key importance in so-called ‘chatty’ distributed applications (applications performance many fine-grained remote method invocations). Network performance is also important in distributed applications moving around big datasets between locations and widely distributed

applications. Memory is especially important when it is a scarce resource (causing systems to swap out memory to the disk). In Java virtual machines the configuration of the memory garbage collection algorithm is important too. The configuration needs to be in line with the type of application, its memory allocation behavior and its workload.

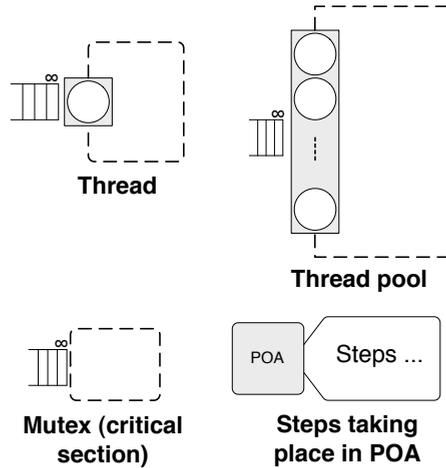


Figure 4.2: Performance model notation

4.3.1 Thread-per-client

The simplest threading strategy available in ORBacus is the thread-per-client model. In the thread-per-client model a receiver thread is created for every incoming connection request of a client. Since clients could ask for multiple connections using the private connection policy or connect to different POA managers in the same server, a better name for this model would be thread-per-connection. This section discusses the performance model of the thread-per-client threading strategy. Figure 4.3 contains the performance model in an extended queuing network notation.

Performance model description

Requests arrive at receiver threads with connection k (of N_{con} connections) arrival rate $\lambda_{i,j,k}$ for POA i and Object j . During request processing the receiver thread is occupied, this resource possession is depicted by a dashed line around the receiver thread logical resource. The thread-per-client model

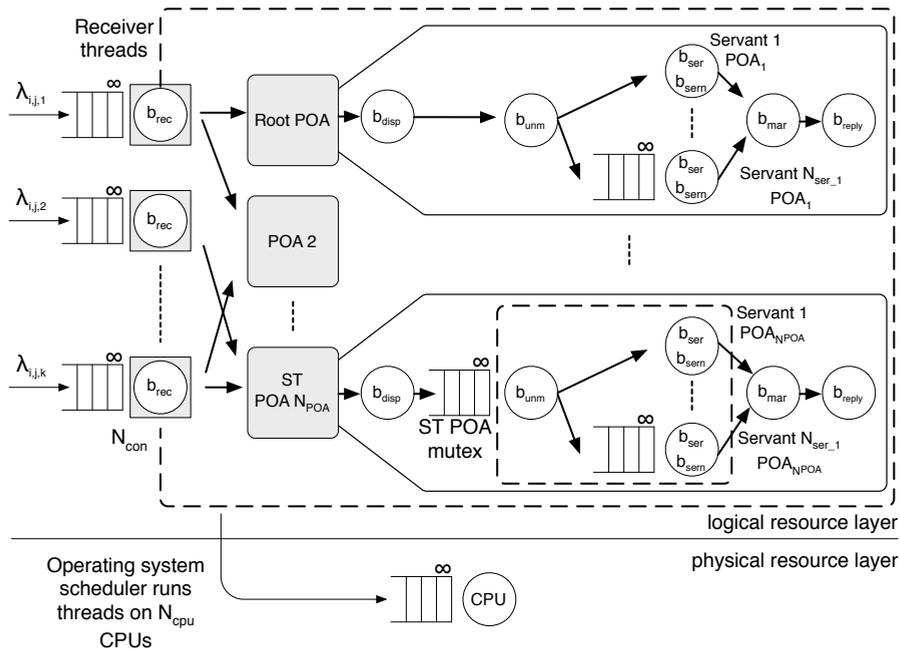


Figure 4.3: Performance model of the thread-per-client strategy

implements the same-thread dispatching model: requests are received and dispatched to servants in the same thread (the receiver thread). While the receiver thread is processing a request, newly incoming requests queue in the FIFO queue.

The receiver thread first receives the request from the network, then it unmarshals the request header and a part of the request body, to obtain the POA and object identifiers. A series of request de-multiplexing steps is needed to locate the target POA, target object, and target method implementation. The POA identifier is used to locate the POA. In the performance model the aforementioned processing steps (including the de-multiplexing step to obtain the target POA, but excluding the other de-multiplexing steps) are modeled by a service demand b_{rec} . The request is forwarded to the located POA, one of N_{poa} POAs. De-multiplexing continues with locating the target object in the POA's active object map, using the earlier obtained target object identifier. Now we have located the CORBA skeleton of the target object. The last de-multiplexing step is to locate the method implementation inside the skeleton. Finally, the request is dispatched to the target object's

skeleton. These two de-multiplexing steps and the dispatching are modeled by a service demand b_{disp} .

The remainder of the request (partial un-marshaling already took place earlier), including the request parameters, is un-marshaled by the skeleton. This un-marshaling step is modeled by a service demand $b_{unm_{i,j}}$. The service demand depends on the amount and type of data that needs to be un-marshaled. The method implementation is invoked with the un-marshaled method parameters. The service demand of the method implementation is modeled by $b_{ser_{i,j}}$ (CPU service demand) and $b_{sern_{i,j}}$ (delay introduced by non-CPU resources, e.g. remote database access or remote procedure calls).

A POA may be configured using the single-thread threading policy, meaning that only one method at a time can be active inside the POA (i.e. in the objects managed by that POA). Access to the objects managed by this POA needs to be serialized. In single-threaded POAs the un-marshaling of the request parameters and the invocation of the method implementation are protected by a mutex, as depicted in the performance model by the ST-POA mutex (a FIFO queue). The critical section, protected by the mutex, is denoted by a dashed line drawn around the involved service centers. The Root POA, the POA created when the ORB starts, is always configured to be multi-threaded (it uses the ORB Controlled Model threading policy by default). The servant implementation may have a critical section too, which can also be modeled by FIFO queues as depicted in the model. The service centers that represent the servants with service demand $b_{ser_{i,j}}$ and delay $b_{sern_{i,j}}$ may be replaced by sub-models if the performance behavior of the servant cannot be captured using these parameters.

After method invocation the return parameters need to be marshaled. This is modeled by a service demand $b_{mar_{i,j}}$. The service demand depends on the amount and type of data that needs to be marshaled. Finally, a reply message is created with the marshaled return parameters. The reply is sent back to the client over the same TCP/IP connection as the request arrived. This last step is modeled by a service demand b_{reply} .

The threads that execute the mentioned steps for each request that is being served, share the physical server resources (such as the CPU, memory and I/O), as indicated in Figure 4.3.

4.3.2 Threaded

The threaded threading strategy is similar to the thread-per-client threading strategy. The only difference is that in the threaded threading strategy there may only be one active request at a time in the servants. Access to the servants is serialized by an ORB-wide mutex. This threading strategy is used by CORBA applications that are not multi-thread aware. For instance, legacy CORBA applications that are developed for single-threaded ORBs can use this threading strategy. Similar to the thread-per-client model, the threaded model also implements the same-thread dispatching model, where the receiver thread both receives requests from clients and dispatches them to the servants.

The performance model of the threaded threading strategy is similar to the one of thread-per-client, except that the mutex for single-threaded POAs is removed (the ORB wide mutex basically makes all POAs single-threaded) and an ORB-wide mutex has been added in the POA model. Figure 4.4 contains the performance model in an extended queuing network notation.

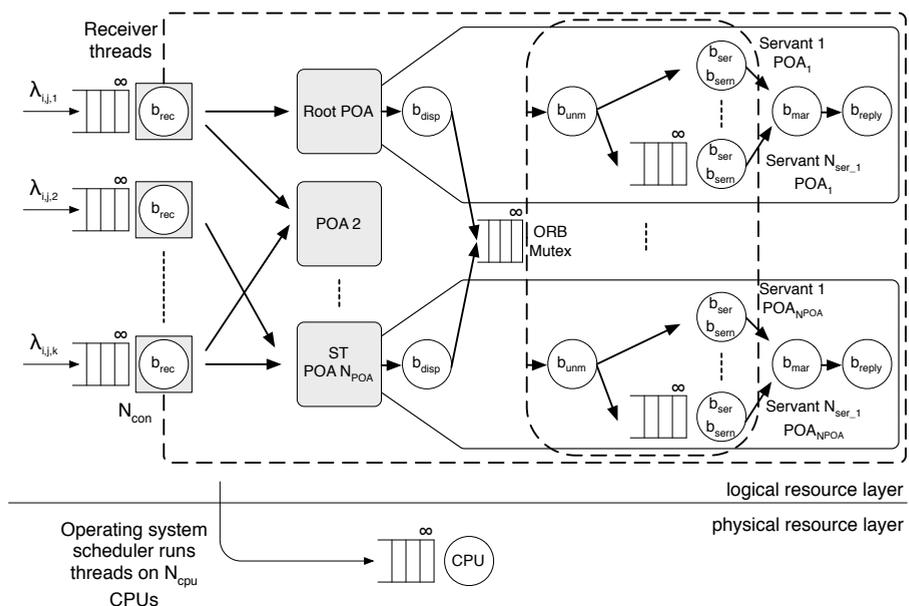


Figure 4.4: Performance model of the threaded strategy

Performance model description

Requests arrive at receiver threads with connection k (of N_{con} connections) at rate $\lambda_{i,j,k}$ for POA i and Object j . During request processing the receiver thread is occupied, this resource possession is depicted by a dashed line around the receiver thread logical resource. The threaded model implements the same-thread dispatching model: requests are received and dispatched to servants in the same thread (the receiver thread). While the receiver thread is processing a request, newly incoming requests queue in the FIFO queue.

The receiver thread first receives the request from the network, then it unmarshals the request header and a part of the request body, to obtain the POA and object identifiers. A series of request de-multiplexing steps is needed to locate the target POA, target object, and target method implementation. The POA identifier is used to locate the POA. In the performance model the aforementioned processing steps (including the de-multiplexing step to obtain the target POA, but excluding the other de-multiplexing steps) are modeled by a service demand b_{rec} . The request is forwarded to the located POA, one of N_{poa} POAs. De-multiplexing continues with locating the target object in the POA's active object map, using the earlier obtained target object identifier. Now we have located the CORBA skeleton of the target object. The last de-multiplexing step is to locate the method implementation inside the skeleton. Finally, the request is dispatched to the target object's skeleton. These two de-multiplexing steps and the dispatching are modeled by a service demand b_{disp} .

The remainder of the request (partial un-marshaling already took place earlier), including the request parameters, is un-marshaled by the skeleton. This un-marshaling step is modeled by a service demand $b_{unm_{i,j}}$. The service demand depends on the amount and type of data that needs to be un-marshaled. The method implementation is invoked with the un-marshaled method parameters. The service demand of the method implementation is modeled by $b_{ser_{i,j}}$ (CPU service demand) and $b_{sern_{i,j}}$ (delay introduced by non-CPU resources).

In the threaded threading strategy only one method at a time can be active inside any POA. Access to all objects needs to be serialized. The un-marshaling of the request parameters and the invocation of the method implementation are protected by a mutex, as depicted in the performance model by the ORB mutex (a FIFO queue). The critical section, protected by the mutex, is denoted by a dashed line drawn around the involved service centers. In this threading strategy the servants don't have critical sections, as they are

single-threaded. As with the thread-per-client model, the service centers that represent the servants with service demand $b_{ser_i,j}$ and delay $b_{sern_i,j}$ may be replaced by sub-models if the performance behavior of the servant cannot be captured using these parameters.

After method invocation the return parameters need to be marshaled. This is modeled by a service demand $b_{mar_i,j}$. The service demand depends on the amount and type of data that needs to be marshaled. Finally, a reply message is created with the marshaled return parameters. The reply is sent back to the client over the same TCP/IP connection as the request arrived. This last step is modeled by a service demand b_{reply} .

4.3.3 Thread-per-request

The thread-per-request threading strategy separates the request receiving and dispatching steps into separate threads. After the receiver thread receives a request from a client, a new thread is spawned to dispatch the request to the servant. Execution of the method implementation takes place in that dispatcher thread. After spawning the dispatcher thread and forwarding the request to that thread, the receiver thread is ready to receive the next request from the client. When the dispatcher thread is done with dispatching the request to the servant and sending the reply back to the client, it kills itself. The advantage of this threading strategy is that multiple requests from the same client can be dispatched to servants concurrently. The disadvantage of this threading strategy is the costs of thread creation and destruction for every request. Especially for requests that require little time to complete, the added overhead of thread creation and destruction is relatively high. However, for requests that take a long time to complete this threading strategy is useful, because it doesn't block further requests by the same client. Because the number of dispatcher threads is unbounded (they are created by receiver threads for incoming requests, as long as the operating system has sufficient resources for the threads) it can lead to an uncontrolled growth of dispatcher threads. When a lot of threads are simultaneously active the memory resources of the machine are drained, and the overhead caused by thread context switches can lead to trashing behavior, where little time is left for actual request processing. Figure 4.5 contains the performance model in an extended queuing network notation.

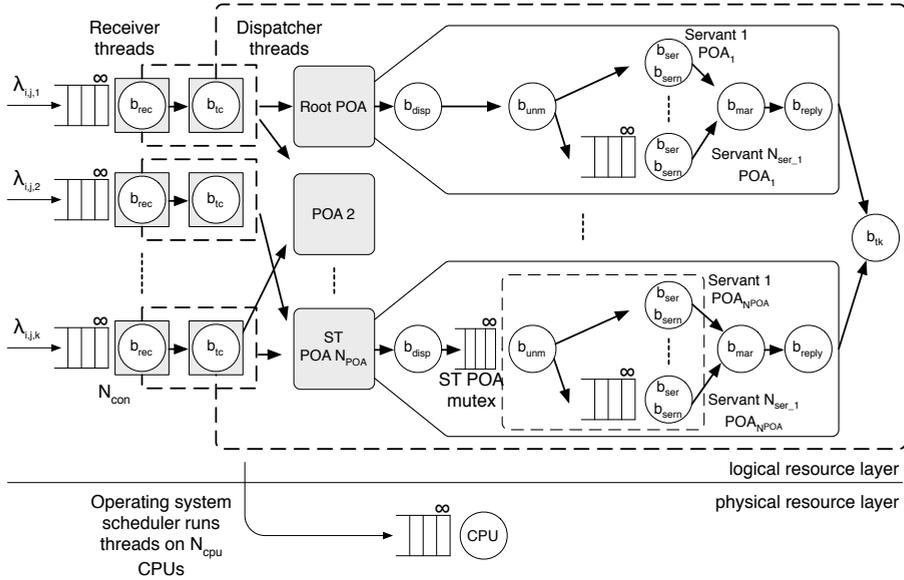


Figure 4.5: Performance model of the thread-per-request strategy

Performance model description

Requests arrive at receiver threads with connection k (of N_{con} connections) with request arrival rate $\lambda_{i,j,k}$ for POA i and Object j . During request processing the receiver thread is occupied, this resource possession is depicted by a dashed line around the receiver thread logical resource. The thread-per-request model implements the separate-thread dispatching model: requests are received in receiver threads and dispatched to servants in other threads, the dispatcher threads. While the receiver thread is receiving a request, newly incoming requests queue in the FIFO queue.

The receiver thread first receives the request from the network, then it unmarshals the request header and a part of the request body, to obtain the POA and object identifiers. A series of request de-multiplexing steps is needed to locate the target POA, target object, and target method implementation. The POA identifier is used to locate the POA. In the performance model the aforementioned processing steps (including the de-multiplexing step to obtain the target POA, but excluding the other de-multiplexing steps) are modeled by a service demand b_{rec} .

Then, a new dispatcher thread is created by the receiver thread. The cost of

thread creation is modeled by the service demand b_{tc} . The dispatcher thread continues processing the request, and the receiver thread is now ready to receive the next request from the queue. After these processing steps the receiver thread is done with the 1st phase of its request processing. The 2nd phase consists of cleaning allocated data-structures and preparations to process the next request. After the 2nd phase the receiver thread will block if no new request is available.

The dispatcher thread forwards the request to the located POA, one of N_{poa} POAs. De-multiplexing continues with locating the target object in the POA's active object map, using the earlier obtained target object identifier. Now we have located the CORBA skeleton of the target object. The last de-multiplexing step is to locate the method implementation inside the skeleton. Finally, the request is dispatched to the target object's skeleton. These two de-multiplexing steps and the dispatching are modeled by a service demand b_{disp} . The remainder of the request (partial un-marshaling already took place earlier), including the request parameters, is un-marshaled by the skeleton. This un-marshaling step is modeled by a service demand $b_{unm_{i,j}}$. The service demand depends on the amount and type of data that needs to be un-marshaled. The method implementation is invoked with the un-marshaled method parameters. The service demand of the method implementation is modeled by $b_{ser_{i,j}}$ (CPU service demand) and $b_{sern_{i,j}}$ (delay introduced by non-CPU resources).

A POA may be configured using the single-thread threading policy, meaning that only one method at a time can be active inside the POA (i.e. in the objects managed by that POA). Access to the objects managed by this POA needs to be serialized. In single-threaded POAs the un-marshaling of the request parameters and the invocation of the method implementation are protected by a mutex, as depicted in the performance model by the ST-POA mutex (a FIFO queue). The critical section, protected by the mutex, is denoted by a dashed line drawn around the involved service centers. The Root POA, the POA created when the ORB starts, is always configured to be multi-threaded (it uses the ORB Controlled Model threading policy by default). The servant implementation may have a critical section too, which can also be modeled by FIFO queues as depicted in the model. The service centers that represent the servants with service demand $b_{ser_{i,j}}$ and delay $b_{sern_{i,j}}$ may be replaced by sub-models if the performance behavior of the servant cannot be captured using these parameters.

After method invocation the return parameters need to be marshaled. This is

modeled by a service demand $b_{mar_{i,j}}$. The service demand depends on the amount and type of data that needs to be marshaled. Finally, a reply message is created with the marshaled return parameters. The reply is sent back to the client over the same TCP/IP connection as the request arrived. This last step is modeled by a service demand b_{reply} .

Now that the request dispatching is done and a reply has been sent to the client, the dispatcher thread kills itself. The cost of killing the thread is modeled by a service demand b_{tk} .

4.3.4 Thread-pool

The thread-pool threading strategy addresses the disadvantages of the thread-per-request threading strategy, while still implementing the separate-thread dispatching model. In the thread-pool model the dispatcher threads are pre-created. Idle dispatcher threads are put in a pool, the dispatcher thread pool. When a request arrives at the receiver thread, it doesn't need to create a new thread for request dispatching, instead the receiver thread forwards the request to the thread pool, where it is queued (in FIFO order). The thread pool request queue is monitored by the idle dispatcher threads. Idle dispatcher threads remove requests from the queue, and dispatch them to servants. When the dispatcher thread is done, it doesn't kill itself, but instead it returns to the thread pool. The thread pool has a fixed size. Therefore the thread-pool threading strategy doesn't suffer from the uncontrolled thread-growth phenomena, unlike the thread-per-request model. Also, since threads are pre-created, the thread creation and destruction costs b_{tc} and b_{tk} of the thread-per-request model are not present here. Figure 4.6 contains the performance model in an extended queuing network notation.

Performance model description

Requests arrive at receiver threads with connection k (of N_{con} connections) with arrival rate $\lambda_{i,j,k}$ for POA i and Object j . During request processing the receiver thread is occupied, this resource possession is depicted by a dashed line around the receiver thread logical resource. The thread-pool model implements the separate-thread dispatching model: requests are received in receiver threads and dispatched to servants in other threads, the dispatcher threads. While the receiver thread is receiving a request, newly incoming requests queue in the FIFO queue.

The receiver thread first receives the request from the network, then it unmarshals the request header and a part of the request body, to obtain the POA

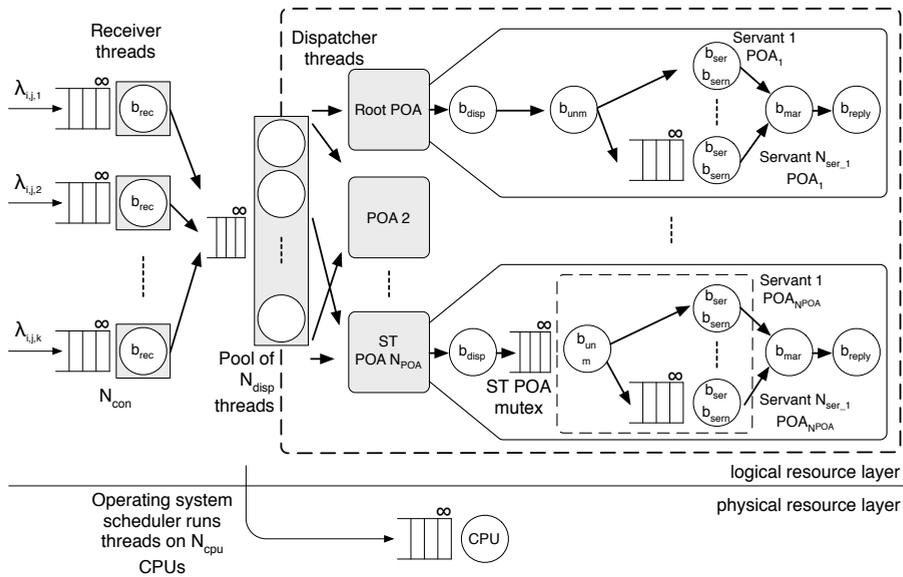


Figure 4.6: Performance model of the thread-pool strategy

and object identifiers. A series of request de-multiplexing steps is needed to locate the target POA, target object, and target method implementation. The POA identifier is used to locate the POA. In the performance model the aforementioned processing steps (including the de-multiplexing step to obtain the target POA, but excluding the other de-multiplexing steps) are modeled by a service demand b_{rec} . Then, the receiver thread enqueues the request in a FIFO request queue in front of the dispatcher thread pool. The dispatcher thread pool contains N_{disp} threads. After enqueueing the request, idle dispatcher threads are signalled that a new request is available in the queue. The request remains in the queue until a dispatcher thread is available to dispatch the request. After these processing steps the receiver thread is done with the 1st phase of its request processing. The 2nd phase consists of cleaning allocated data-structures and preparations to process the next request. After the 2nd phase the receiver thread will block if no new request is available.

The dispatcher thread forwards the request to the located POA, one of N_{poa} POAs. De-multiplexing continues with locating the target object in the POA's active object map, using the earlier obtained target object identifier. Now we have located the CORBA skeleton of the target object. The last de-

multiplexing step is to locate the method implementation inside the skeleton. Finally, the request is dispatched to the target object's skeleton. These two de-multiplexing steps and the dispatching are modeled by a service demand b_{disp} .

The remainder of the request (partial un-marshaling already took place earlier), including the request parameters, is un-marshaled by the skeleton. This un-marshaling step is modeled by a service demand $b_{unm_{i,j}}$. The service demand depends on the amount and type of data that needs to be un-marshaled. The method implementation is invoked with the un-marshaled method parameters. The service demand of the method implementation is modeled by $b_{ser_{i,j}}$ (CPU service demand) and $b_{sern_{i,j}}$ (delay introduced by non-CPU resources, such as remote database access or remote procedure calls).

A POA may be configured using the single-thread threading policy, meaning that only one method at a time can be active inside the POA (i.e. in the objects managed by that POA). Access to the objects managed by this POA needs to be serialized. In single-threaded POAs the un-marshaling of the request parameters and the invocation of the method implementation are protected by a mutex, as depicted in the performance model by the ST-POA mutex (a FIFO queue). The critical section, protected by the mutex, is denoted by a dashed line drawn around the involved service centers. The Root POA, the POA created when the ORB starts, is always configured to be multi-threaded (it uses the ORB Controlled Model threading policy by default). The servant implementation may have a critical section too, which can also be modeled by FIFO queues as depicted in the model. The service centers that represent the servants with service demand $b_{ser_{i,j}}$ and delay $b_{sern_{i,j}}$ may be replaced by sub-models if the performance behavior of the servant cannot be captured using these parameters.

After method invocation the return parameters need to be marshaled. This is modeled by a service demand $b_{mar_{i,j}}$. The service demand depends on the amount and type of data that needs to be marshaled. Finally, a reply message is created with the marshaled return parameters. The reply is sent back to the client over the same TCP/IP connection as the request arrived. This last step is modeled by a service demand b_{reply} .

Now that the request dispatching is done and a reply has been sent to the client, the dispatcher thread returns to the thread pool and is ready to process the next request.

4.4 Workload generation

To aid in our experiments, we developed a synthetic workload generator so that we could automate performance experiments with different scenarios. The workload generator consists of a client and server application.

The server side application offers the following scenario configuration options:

- Specification of the POA hierarchy, including POA managers and POA policies (e.g. single-threaded POAs).
- Deployment of objects on the specified POAs.
- Service demands for methods in the object implementation. Both CPU time usage and waiting time can be described. The CPU time can be used to work that is done by the object implementation. The waiting time can be used to simulate that the object implementation is waiting for an external entity, for instance a query to a remote SQL database. Currently two distributions are supported, exponential service demands and deterministic service demands.
- Configuration options for the ORB, for instance which threading model to use or connection reuse policies.

The client-side application executes a given workload on the server application. The workload description consists of a collection of arrival processes. An arrival process description consists of:

- The total number of requests to generate.
- The targets of the arrival process. A description of a target consists of the name of the remote object, the name of the method to invoke, and (if applicable) requests parameters (payload). If an arrival process has multiple targets, then by default the requests are equally distributed over the targets. It is possible, however, to specify routing probabilities for each target.
- Request inter-arrival times can be defined using the deterministic and exponential distribution (Poisson inter-arrivals). Support for other distributions can easily be added. Requests are generated regardless of the completion of previous requests; i.e. a transaction-class workload, modeled as open arrivals in queuing models.

- Another option is to use a closed arrival process. Requests are sent one after another (at most one outstanding request per arrival process definition) optionally with delays between requests (representing user thinking time); i.e. a new request is sent after the reply to the previous request is received by the client.

Support for non-synthetic workload distributions can be easily added to this CORBA middleware workload generator. An example of such non-synthetic workload is trace-driven load generation using logged workload information from a production environment.

Performance experiments often iterate one or more parameters in the scenario. For instance, a series of experiments can be performed to study the effect of an increasing request rate or an increasing number of clients on the mean response time of requests. We use scripts that iterate these parameters and instantiate workload scenarios templates using the parameter values.

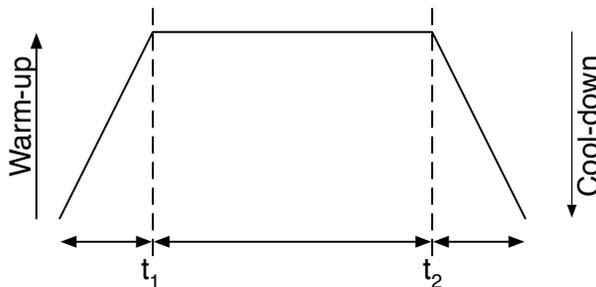


Figure 4.7: Warm-up and cool-down

We illustrate the warm-up and cool-down issues in Figure 4.7. During the warm-up phase the system initializes itself, the Java virtual machine compiles any code hot spots to machine code (just-in-time compilation), caches are filled, etc. The system warm-up is completed when the system has adapted to the given workload, in Figure 4.7 we denote this by t_1 . At time t_2 the workload generator has sent all requests specified in the workload definitions and stops generating new requests. The system may still be working on requests and the load generator waits for those to complete. Meanwhile the load of the system goes down as fewer requests are being processed. This has an impact on the measured performance, e.g. the response times will drop since there are fewer requests inside the system sharing the resources.

Request measurements before t_1 and after t_2 should not be used as they are influenced by warm-up and cool-down effects. The workload generator has no features of its own to deal with system warm-up and cool-down issues. This is left to the performance analyst. In our experiments we filter the requests in the scripts that process all the measurement results.

4.5 Throughput comparison of the threading strategies

In this section we assess and explain the impact of threading strategies on the server performance by doing an experimental comparison of the thread-per-client, threaded, thread-per-request and thread-pool strategies.

4.5.1 Experimental setup

We perform a series of experiments using a closed arrival process with an increasing number of clients. Initially each client sends a single request and waits for the reply. Then a new request is sent after the reply to the previous request is received by the client. This way each client has at most one outstanding request. The reason for using a closed arrival process, instead of an open arrival process, is twofold. First of all, in a set-up with open arrivals TCP/IP sessions have to be set-up and terminated during the experiments. This overhead will degrade the end-to-end performance, and the performance of the threaded and thread-per-client strategy will be affected most. Because we are focusing on a fair comparison of server performance we want to exclude this effect from our results. Secondly, since we are comparing the performance of several threading strategies, the most interesting results are regarding the performance under 'maximum load' instead of under light load. With the closed loop arrival process used in this experiment, the 'maximum load' for each strategy is realized automatically.

Each client has its own TCP/IP connection to the server ORB, and thus its own receiver thread on the server. Our test-bed consists of two machines: Utip267 and Utip442. Utip267 is a Pentium IV 1.7 GHz with 512 MB of memory. Utip442 is a Pentium III 550 MHz with 256 MB of memory. In these experiments Utip442 acts as the CORBA server and Utip267 as the CORBA client. Notice that we used the faster machine as the client in order to make sure that the client does not become the bottleneck in the experiments. In particular, for all results presented below we verified that the request rate generated by the client was at least enough to keep the server busy at all

times (i.e. the client is not the bottleneck). Both machines run the Linux 2.4.19 operating system and the Sun Java 2 standard edition v1.4.1. The Java virtual machine is configured with default garbage collection settings and without run-time pre-compilation optimization features. The CORBA implementation we use in this example is IONA ORBacus/Java 4.1.1. The thread pools used in the experiments with the thread-pool strategy hold a number of threads equal to the number of clients (i.e. it varies with the number of clients). The following is a fragment of the IDL definitions used in the experiments.

```
interface PerformanceTest
{
    long doSomeWork();
};
```

Listing 4.1: IDL definitions for the throughput experiments

The `doSomeWork` method executes a configured work-load on the system. In the experiments we use three workload cases. The 1 ms CPU demand scenario represents the CPU processing cost of a simple method. The 5 ms CPU demand represents the CPU processing costs of a scenario with a more complex method (a CPU bound application). Finally, the 50 ms delay (not CPU processing time) represents the delay induced from a simple SQL query on a database server running on another machine (an I/O bound, database driven application). All service demands and delays are configured to have an exponential distribution. We run the experiment with 2, 4, 8, 16, 32 and 64 clients. Some experiments are also executed with 128 clients, depending on the CPU utilization at 64 clients. Each client executes a work-load of 200 requests on the server. We also have configured a minimum duration of 45 seconds for each experiment, so that we get enough measurements for runs with a small number of clients.

4.5.2 Experimental results

In all experiments the thread-per-client threading strategy is expected to be the most efficient, since we use single-threaded clients executing one blocking request at a time. In this scenario it doesn't make sense to release the receiver thread for processing forthcoming requests, since they won't arrive because the client is single threaded and blocking until it receives a reply for the current outstanding request. Therefore, the results for the thread-per-client strategy can be regarded as best case results. We emphasize that this observation is based on our choice to use a closed arrival process.

1 ms servant CPU demand

Figure 4.8 shows the throughput (in number of requests per second) as a function of the number of clients, for the different threading strategies. The results demonstrate that the thread-per-client and thread-pool threading strategies perform best in this experiment and scale well with the number of connected clients. The slow decrease of the throughput presented in Figure 4.8 is due to the fact that the CPU service time per request increases slightly from roughly 1.8 ms for 2 clients to 2.3 ms for 64 clients. This increase in CPU service time is most likely due to additional context switching activity.

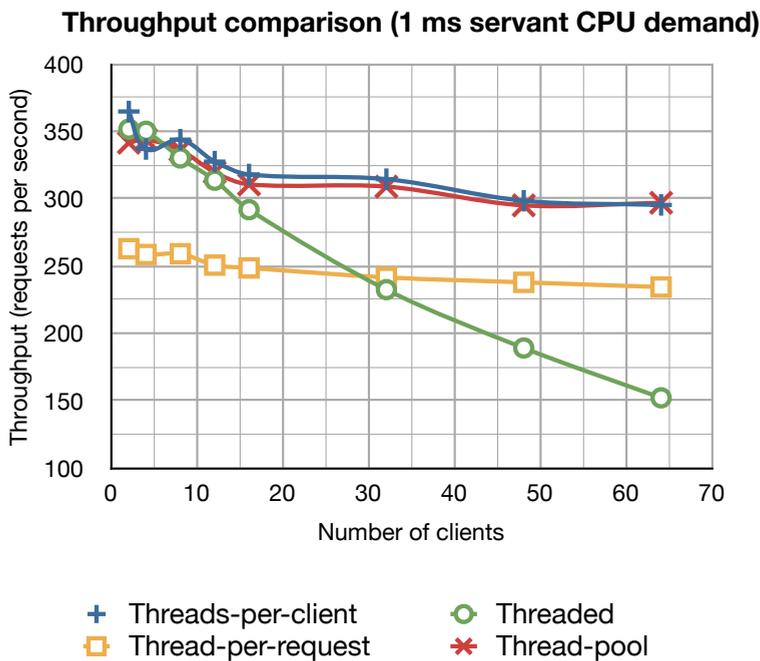


Figure 4.8: Measured throughputs with 1 ms servant CPU demand

The thread-per-request strategy suffers from high thread creation and destruction costs, especially compared to the small service demand of 1 ms CPU time. This is the reason why the throughput obtained with the thread-per-request strategy is lower than the throughput for the thread-per-client and thread-pool strategy. For the threaded strategy the ORB mutex turns out to be a bottleneck, especially with a large number of clients. In particular, the locking activity strongly increases the processing time. For 2 clients the

CPU service time per request is approximately equal to 1.8 ms, while for 64 clients the CPU times have increased to 5.4 ms. The increase of CPU times causes the linear decrease of throughput for the threaded strategy shown in Figure 4.8.

5 ms servant CPU demand

Figure 4.9 contains the throughputs for the different threading strategies and the scenario with 5 ms CPU servant demand. First of all, note that the throughputs are significantly lower than the throughputs for the previous case. Of course, this is due to the fact that the servant is now more CPU demanding. Similar as for the previous scenario the thread-per-client and thread-pool perform best of the four threading strategies.

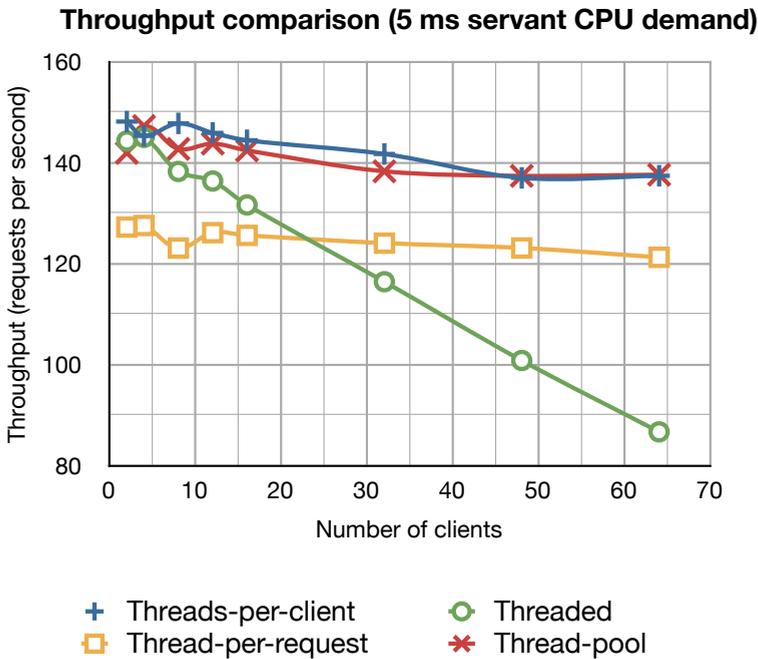


Figure 4.9: Measured throughputs with 5 ms servant CPU demand

The throughput for the thread-per-request strategy remains smaller than the throughput for the thread-per-client and thread-pool strategy, but relatively the thread-per-request strategy performs better in this scenario. For a scenario with 1 ms CPU servant demand the throughput for the thread-

per-request strategy was between 75 and 80% of the throughput for the thread-per-client strategy. For the 5 ms CPU servant demand the relative throughput increases to between 85 and 90%. This relative improvement of the thread-per-request strategy is due to the fact that the thread creation and destruction overhead becomes less, relative to the increased servant CPU demand. For the threaded strategy we observe the same phenomenon as for the previous scenario. For a small number of clients the throughput performance is comparable to the throughput achieved with the thread-per-client and thread-pool strategy. However, for a large number of clients the throughput performance becomes significantly worse. Again this is caused by the additional CPU demand for handling locking contention for the ORB mutex of the threaded threading strategy.

50 ms servant delay

Figure 4.10 contains the throughput comparison for the thread-per-client, thread-per-request, thread-pool and threaded threading strategies, for 50 ms 'sleep time' at the servant. The performance of thread-per-client and thread-pool is the same again. Observe that the throughput curves are different from the previous throughput curves. In particular, for a low number of clients the throughput increases linear with the number of clients. This effect is due to the servant 'sleep time' of 50 ms. Observe that the sleep time causes that the request 'loop time' (i.e. the elapsed time between two consecutive arrivals of a request at the server) is at least 50 ms and this provides an upper bound on the maximum throughput per client, of $1 / 0.05 = 20$ requests per second. Then, for n clients the maximum achievable throughput equals $n \times 20$ requests per second. For a large number of clients the CPU becomes the bottleneck. For the thread-per-client and thread-pool strategy this point is reached at approximately 30 clients. For the more CPU demanding thread-per-request strategy this point is reached around 20 clients.

For the threaded strategy we observe a completely different throughput performance. Again, this is due to the ORB mutex, which does not allow the servant to be invoked by more than one request at the time. In combination with the 'loop time' observation above, it follows that the servant can never handle more than 20 requests per second. And this exactly corresponds to the throughput results shown in Figure 4.10.

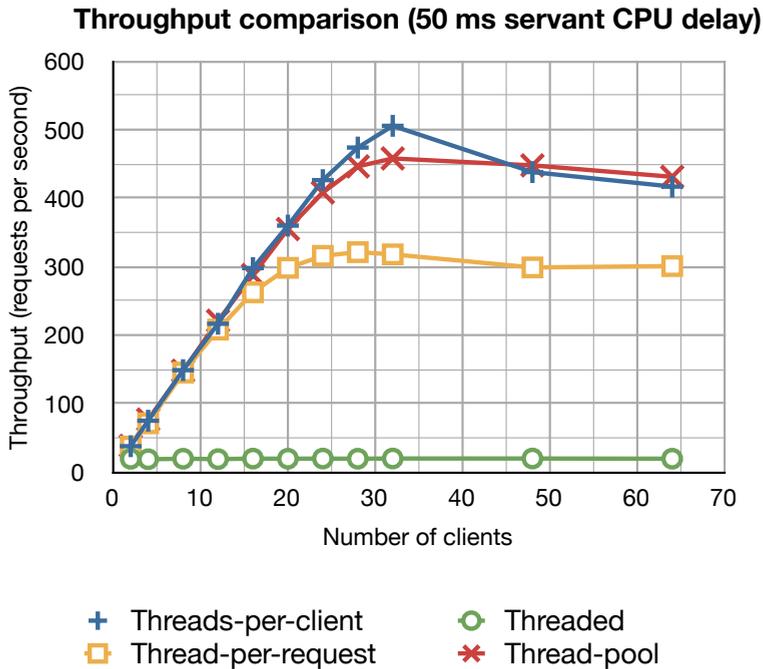


Figure 4.10: Measured throughputs with 50 ms servant delay

4.5.3 Summary

To summarize, the thread-per-client threading strategy is the best performer. This was to be expected since the work-load of clients executing one request at a time perfectly fits that threading strategy. The thread-pool is overall the second best performer. Contrary to the thread-per-client threading strategy, the thread-pool model also copes with multi-threaded clients, which invoke more than one method at a time over a client-server connection. In the thread-per-client threading strategy the receiver thread is not separated from the method dispatching thread, i.e. the server ORB cannot handle requests coming from the same client concurrently. The thread-pool model is a good choice for applications that have multi-threaded clients.

When designing and implementing a CORBA application, the choice of which threading strategy to use is an important issue. In many cases the requirements of an application already point to certain threading strategies. For instance,

- Legacy applications would use the threaded model if they cannot handle concurrent requests.
- Applications that want to restrict the number of simultaneous method invocations by a client can use thread-per-client, so that only one request at a time is handled for each client.
- Thread-per-client or thread-pool would be used for applications with multi-threaded clients.
- Applications that will likely suffer from uncontrolled thread growth, for instance, when bursts of requests are expected at times, can use thread-pool with a number of pre-allocated threads.
- Applications with CPU intensive servants will likely want to limit the number of simultaneously active dispatching threads, for instance for QoS reasons. These applications would use the thread-pool threading strategy. The thread-per-client threading strategy could also be used to this end, but the number of clients should be bounded.

Applications whose client-side is not multi-threaded will not benefit from the thread-per-request and thread-pool threading strategies. The server-ORB should be deployed with the thread-per-client threading strategy if only single-threaded clients connected, or clients that only invoke one (blocking) method at a time. Single-threaded clients that use non-blocking requests (oneway asynchronous or deferred synchronous), could still benefit from the concurrency of thread-per-request or thread-pool.

4.6 Impact of marshaling

In this section we investigate the overhead of the marshaling and un-marshaling actions in CORBA method invocation requests. Marshaling and un-marshaling is the conversion to and from a common data representation by the ORB. This conversion to a common data representation is needed to ensure interoperability between ORBs running on different machine architectures. The marshaling and un-marshaling actions part of the request processing can take a significant portion of the request processing time [15]. Previous research [4] concluded that the overhead is linearly dependent of the size of the data that is marshaled and un-marshaled. [4] also concluded that the data-type wasn't a major factor in the overhead. This section presents results from performance experiments that quantify the overhead of marshaling

and un-marshaling for various data-types and various data sizes. We found that garbage collection is also an important factor to consider, so we have included garbage collection statistics in the experiments results too.

4.6.1 Experimental setup

These experiments use the same test-bed as described in Section 4.5.1. Utip442 acts as the CORBA server and Utip267 as the CORBA client, running the workload generator. We have configured our CORBA workload generator to send one request after another (i.e. there is at most one active CORBA method invocation). In the experiments we used the data-types octet, char, long, float and string. The following is a fragment of the IDL definitions used in the experiments.

```
interface PerformanceTest
{
    typedef sequence<octet> OctetSeqType;
    typedef sequence<char> CharSeqType;
    typedef sequence<long> LongSeqType;
    typedef sequence<float> FloatSeqType;

    void marshalTest_in_octetSeq
        (in long in_octetSeq_size ,
         in OctetSeqType in_octetSeq);
    void marshalTest_in_charSeq
        (in long in_charSeq_size ,
         in CharSeqType in_charSeq);
    void marshalTest_in_longSeq
        (in long in_longSeq_size ,
         in LongSeqType in_longSeq);
    void marshalTest_in_floatSeq
        (in long in_floatSeq_size ,
         in FloatSeqType in_floatSeq);
    void marshalTest_in_string
        (in long in_string_size ,
         in string in_string);
    void marshalTest_out_octetSeq
        (in long in_octetSeq_size ,
         out OctetSeqType out_octetSeq);
    void marshalTest_out_charSeq
        (in long in_charSeq_size ,
         out CharSeqType out_charSeq);
    void marshalTest_out_longSeq
        (in long in_longSeq_size ,
```

```

    out LongSeqType      out_longSeq);
void marshalTest_out_floatSeq
(in   long              in_floatSeq_size ,
 out  FloatSeqType     out_floatSeq);
void marshalTest_out_string
(in   long              in_string_size ,
 out  string            out_string);
};

```

Listing 4.2: IDL definitions for the marshaling experiments

The octet, char, long and float data is sent in sequences (arrays), except for the string data-type which is already an array of character data. The following sequence sizes are used: 0, 1024, 2048, 4096, 8192, 16384, 32768, 40960, 65536, 73728, and 98304.

4.6.2 Experimental results

Processing times for marshaling and un-marshaling

The following table, Table 4.2, contains CPU processing times for marshaling and un-marshaling of the octet, char, string, long, and float basic types.

	Octet		Char		String		Long		Float	
Size	Mars.	Un-m.	Mars.	Un-m.	Mars.	Un-m.	Mars.	Un-m.	Mars.	Un-m.
0	0.0073	0.0093	0.0072	0.0093	0.0364	0.0312	0.0072	0.0094	0.0072	0.0092
1024	0.0270	0.0173	0.5142	0.4090	0.5734	0.4529	1.2598	1.2408	1.6914	1.7549
2048	0.0318	0.0206	1.0065	0.8045	1.1076	0.8773	2.4957	2.4686	3.3626	3.4876
4096	0.0461	0.0268	1.9921	1.5966	2.1780	1.7333	4.9760	4.9286	6.6142	6.9525
8192	0.0730	0.0440	3.9577	3.1762	4.3450	3.4705	9.9349	9.8451	13.3899	13.8978
16384	0.1483	0.0857	7.9300	6.3422	8.6376	6.9440	19.2034	20.4090	26.6143	28.0180
32768	0.2736	0.1575	15.8735	12.6667	16.1777	13.7096	38.3185	40.8920	53.5810	56.0637
65536	0.5441	0.3277	31.5984	25.3374	32.5772	30.8475	76.9825	82.0031	108.4025	112.3944

Table 4.2: Processing times for marshaling and un-marshaling

All experiments for both marshaling and un-marshaling show a linear dependency between the amount of data which needs to be marshaled or un-marshaled and the amount of CPU processing time required for marshaling and un-marshaling. From the data we can also conclude that the processing costs for marshaling are comparable to those of un-marshaling, for the same data type and amount. When looking at the individual types, octet has the smallest marshaling overhead; this is because octets are transferred as is, no conversion is done. Sequences of characters and strings have comparable marshaling costs. The more complex basic types long and float have the largest marshaling costs. Finally, we can observe that the costs per

byte for all types except octet are reasonably similar (the long and float are 4 bytes, char 1 byte, and a string is basically an array of char).

Garbage collection

In these experiments we used the default garbage collector, the stop-the-world collector. We also used the default garbage collector settings, such as the size of the heap. The results show increased garbage collection activity with higher sequence sizes. The garbage collection cycles can have a major influence on the completion times of requests. Especially with the stop-the-world garbage collector in current Java virtual machines, garbage collection is a QoS issue: garbage collection can cause large variances between completion times. Modern virtual machines include better garbage collection algorithms, such as incremental garbage collection. With incremental garbage collection the virtual machine is not stopped during garbage collection cycles, and the cycles are more evenly distributed during program execution. Especially for programs that more stringent QoS constraints than best-effort, we recommend the use of more advanced garbage collectors. We can also observe differences in the garbage collector times, some cycles take around 30 msec while other cycles take around 130 msec. The stop-the-world collector uses 2 different object groups: young and old generation objects. Objects that exist for a longer duration are promoted to the old generation group. Garbage collection cycles for the young generation objects takes less time than those for the old generation. Also, garbage collection cycles for the old generation occur less frequently.

4.7 Modeling the thread scheduling

The processing steps performed by the threads in the performance models described in the previous sections share the various hardware resources, among them the central processing unit (CPU). In this section we will discuss how this CPU sharing could be implemented in the performance models. First, by employing the often used processor sharing (PS) scheduling discipline. Second, by modeling the scheduling in more detail. In our case, by exploring Linux thread scheduling.

4.7.1 Processor sharing

The most straightforward way to describe the sharing of the CPU resource by the threads is to model the CPU resource by a queuing node with the

processor sharing (PS) scheduling discipline. That is, if at some point in time there are in total N receiver and dispatcher threads active, then each of them receives a fraction of $1/N$ of the available processor capacity.

4.7.2 Linux 2.4 thread priority scheduling

In modern operating systems the threads are scheduled according to some priority mechanism. For instance, Linux v2.4 schedules thread execution using a time sharing scheduler with variable quantum of 10 – 110 milliseconds (depending on the time-slice a thread has left to spend) [6]. The scheduling order of threads depends on the time-slice threads have left: when the kernel needs to determine the next thread to schedule, runnable threads with a higher time-slice have higher priority. When there are no runnable (non-blocking) threads left with a time-slice larger than zero, the time-slices of all threads are recalculated. Threads keep half of their remaining time-slice from the last scheduler round and some constant amount of time-slice is added. By allowing threads to keep half of their remaining time-slice, Linux favors threads that don't spend a lot of time on the CPU (i.e. sleeping / blocking threads). In other words, Linux favors I/O bound threads over CPU bound threads. Every 10 milliseconds the kernel decrements the time-slice of the running thread, this happens during the system's timer-interrupt handling.

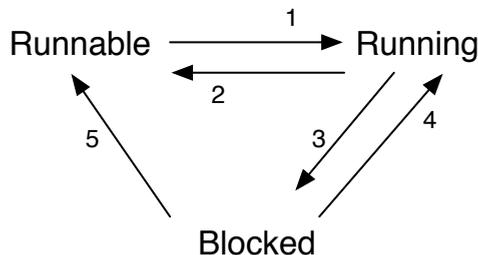


Figure 4.11: The three main thread states

In Figure 4.11 the three most important states a thread can be in are depicted: running, runnable, or blocked. A running thread is the thread that is currently using the CPU. Runnable threads are ready to use the CPU, but have to wait for the running thread to switch context. Blocked threads are threads that are waiting for some resource to become available, for instance a packet from the network, or a request in a queue (e.g. the dispatcher thread-pool's queue).

A thread's state changes as a result of events that occur in the system. In Figure 4.11 there are five of such events:

1. A currently runnable thread becomes the running thread. This happens when the current running thread runs out of time-slice or blocks for some resource. The runnable thread with the highest remaining time-slice becomes the running thread. The former running thread changes to runnable state if the time-slice ran out, or blocked if it is blocking for some resource. When no runnable threads with a time-slice larger than zero are left, the scheduler recalculates all time-slices and a new scheduler round is initiated, as discussed above.
2. A currently running thread changes to runnable state. This happens when a thread with a higher remaining time-slice than the running thread un-blocks. The un-blocked thread becomes the running thread. This state change also occurs when the running thread runs out of time-slice (see 1.).
3. A currently running thread blocks for some resource. One of the runnable threads becomes the new running thread (see 1.).
4. A blocked thread un-blocks because some resource has become available and the un-blocked thread has a higher remaining time-slice than the current running thread and all the runnable threads.
5. A blocked thread un-blocks because some resource has become available, but the un-blocked thread does not have a higher remaining time-slice than the current running thread and all the runnable threads. Note that this state change (from blocked to runnable) may still trigger another runnable thread to become the running thread, since after un-blocking the scheduler checks if there is a runnable thread with a higher time-slice remaining than the current running thread (the current running thread's time-slice may have been decremented one or more times when timer interrupts occur).

As mentioned above, Linux, and in fact many modern operating systems, including many UNIX systems [57] and Microsoft Windows [46], calculate thread scheduling priorities based on their CPU usage. This influences the scheduling behavior of the middleware's receiver and dispatcher threads. For example, if we have one receiver thread which uses 1 millisecond of CPU time to process an incoming request and one dispatcher thread which

uses 5 milliseconds of CPU time per request, the dispatcher thread uses 5 times as much CPU time. This means that if such a middleware server is under a continuous load the scheduling priority of the receiver thread would be 5 times higher than the dispatcher thread's priority. However, when the middleware server is configured with 5 dispatcher threads, the requests are spread over those 5 threads and the priorities of the receiver thread and a dispatcher thread would be about the same. When the middleware has 10 dispatcher threads, the priorities of those dispatcher threads would be higher than the priority of the receiver thread. In the new simulation results in Section 5.3.3 we can see this behavior clearly: around 5 dispatcher threads the lines of the receiver and dispatcher thread's queuing times cross each other.

Note that without sufficient load the middleware threads remaining time-slices would converge to the maximum in the Linux kernel (which is 110 milliseconds) and their priority would be the same, since the threads would mostly be blocked (and hence can save up time-slice).

4.8 Summary

In this chapter we have explored threading in CORBA middleware implementations. We have combined our insight in CORBA object middleware, Java and multi-threading behavior to construct performance models for four often implemented threading strategies

In the next chapter we will discuss the implementation of these performance models using the Extend simulation environment [33] and our own performance simulation tool for distributed applications. We validate the performance models by comparing model-based results with real-world measurements.

PERFORMANCE MODEL VALIDATION

In this chapter we describe simulations implementing the CORBA performance models along with their experimental validation.

This chapter is structured as follows. Section 5.1 discusses how we simulate our performance models. Section 5.2 introduces our performance simulation engine for distributed applications. Section 5.3 validates the performance model for the thread-pool scheduling strategy for an increasing number of dispatcher threads. Section 5.4 validates our performance models for an increasing number of clients. Section 5.5 summarizes this chapter.

5.1 Performance model implementation

We have implemented the thread-pool performance model described in section 4.3.4 in the Extend [33] simulation environment. The Extend environment can be used to implement models without having to use an advanced simulation programming language, though under water there still is a simulation programming language. Models are constructed by connecting pre-built model building blocks together on a grid, using the Extend GUI. The user is not restricted to the pre-built building blocks. It is possible to add new building blocks, using the built-in simulation language ModL. Extend has a large library with pre-built building blocks, for instance blocks that implement the FIFO (first-in first-out) and PS (processor sharing) service

disciplines. Extend models can be expressed in hierarchical manner, where blocks implement sub-models. Each block can have various parameters. The model parameter values can be specified in the notebook, for instance the inter-arrival times between requests, and the number of threads in the dispatcher thread pool.

For generating simulation results, Extend offers various building blocks to calculate results and report graphs. Using these blocks the model implementer can retrieve the required performance measures from the model, such as response times, throughput, queuing times, and resource utilizations.

Our first version of the middleware performance model in Extend used the processor sharing discipline, described in section 4.7.1. While the predictions by this model were good for many scenarios, it would not predict queuing times accurately in all cases. We have replaced the processor sharing discipline with a fixed priority scheduler in the Extend model. The priorities are calculated based on the CPU service demands of threads for each request, before the simulation starts. Clearly, using fixed priorities works best if the CPU service times of the threads in the system are deterministically distributed. Variation in the workload and service demands may lead to less accurate results, since the fixed priorities will represent the average case.

The implementation of the performance model in Extend has been a bug-ridden process. At various occasions we found bugs or quirks in the Extend simulation library that caused our performance model to generate invalid results. In our first performance model we had problems with the implementation of processor sharing scheduling in Extend. During the development of the second model, with fixed priority scheduling, we had problems with the implementation of priority queues in Extend.

Our problems with the Extend simulation tool and our desire to experiment with various scheduling algorithms caused us to develop a custom discrete event simulation engine. The next section describes this simulation engine.

5.2 The Distributed Applications Performance Simulator

DAPS (Distributed Applications Performance Simulator) is a discrete event simulator, especially targeted at distributed multi-threaded applications. The DAPS simulation engine offers the following building blocks to the modeler: schedulers, threads, queues, mutexes, network delay servers, programs, mutexes, and open arrival processes.

A *scheduler* represents the heart of a machine. It schedules threads for execution on the CPU using some scheduler algorithm. The simulator supports the processor sharing scheduler, the round-robin scheduler, and the Linux 2.4 scheduler (an abstraction of the scheduler present in the Linux 2.4 operating system). Each *thread* handles work that is en-queued in its source *queue*. These queues can be *shared* by multiple threads (implementing threads pools) or be *private* to a single thread (implementing for instance connection accepting threads). The simulator offers a simple *program* building block to express actions to be performed by the threads. These actions include using a certain amount of CPU time, sleeping (waiting), obtaining a lock (mutual exclusion), performing a *nested call* on another thread, and performing a *forwarding call* on another thread. The difference between a nested and forwarding call is that with nested calls, the caller waits for the callee to finish execution. After execution of the nested call the caller continues work (meanwhile the caller thread blocks). With forwarding calls the responsibility to returning a reply to the client is forwarded to the callee and the caller can continue execution without having to wait for the callee to finish execution. An example of a forwarding call is the receiver thread in our middleware model. It moves the request to the dispatcher thread pool queue, after en-queuing the request the receiver thread is ready to receive the next request. The responsibility of sending a reply to the client, is forwarded to the dispatcher thread. The *mutex* building block can be used by programs to simulate mutual exclusion. Locking contention results in a blocked thread. Releasing a lock causes blocked threads (contending for the lock) to become runnable again. Lastly, requests can be generated by the *open arrival process* building block.

Figure 5.1 depicts the performance model structure in DAPS. The bottom layer consists of a discrete event simulation. On top of this simulation various schedulers can be instantiated, representing various machines used by the distributed application. For instance, the physical resource layer of our performance model in Figure 4.6, consisting of a thread scheduler, is represented here. Finally, the application performance behavior, the logical resource layer in Figure 4.6, consisting of logical resources and the interactions/dependencies between them, is modeled on top of the schedulers.

The DAPS simulator has some limitations. Firstly, the simulator cannot be used to simulate multi-processor machines. Secondly, the network delay 'server' implementation is too simplistic to accurately model TCP response times for RPC and HTTP traffic. Thirdly, closed arrival processes have not been implemented yet. Fourth, resource constraints, such as maximum

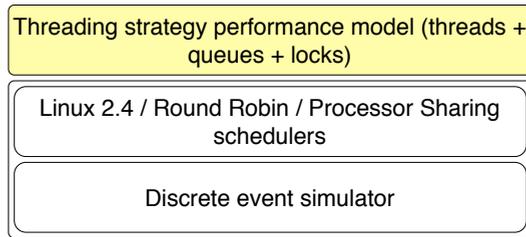


Figure 5.1: Modeling in DAPS

number of threads to allocate and maximum number of TCP connections, have not been implemented in this version of the simulator.

The next section discusses the validation of the performance model for the thread-pool scheduling discipline, for both the Extend PS & fixed priority model implementations and the implementation using DAPS.

5.3 Validation of the thread-pool strategy for an increasing number of dispatchers

In this section we validate the performance model of the thread-pool threading strategy.

5.3.1 Experimental setup

Our test lab consists of 2 machines interconnected using a local network. The server machine is a Pentium III 550 MHz with 256 MB RAM. The client machine is a Pentium IV 1.7 GHz with 512 MB RAM. Both machines run the Linux v2.4 operating system and the Java 2 standard edition v1.4.1. For this experiment we disabled priority scheduling of processes on the Linux machines and used high-resolution timers to generate accurate arrival processes. The CORBA implementation we use is ORBacus 4.1.1 by IONA Technologies [ORBacus00]. In the experimental setup one target object, managed by the Root-POA, is instantiated in each scenario. The client machine runs a synthetic workload generator that produces workload for the CORBA implementation running on the server machine.

To obtain measurement data of the CORBA server, we used our Java Performance Monitoring Toolkit (JPMT). Amongst other measures, we monitored

the following performance data during the experiments:

- Queuing times before the receiver thread and the dispatcher thread-pool.
- CPU usage of the receiver and dispatcher threads.
- Holding times of the receiver and dispatcher threads, plus 1st and 2nd phase times of receiver and dispatcher threads. The holding time is the total time a thread is busy processing a request. The holding time consists of a 1st phase and a 2nd phase. The 1st phase is the actual processing of the request by the thread. This phase ends with the receiver or dispatcher thread moving the request to a queue or another thread, or sending a reply to the client. The 2nd phase is the cleaning up afterwards (for instance data-structures) and getting ready to process the next request.
- CPU utilization, so that we know our experiments do not saturate the CPU resource, thus making the system unstable.

5.3.2 Experiment results

We configure the CPU service demand of the target object to be 5 milliseconds with a deterministic distribution. The client ORB will generate a workload of 15000 requests using a Poisson process, via one connection. The arrival-rate for the Poisson process is 1 request per 10 milliseconds. The number of dispatcher threads available in the server ORB varies per experiment, between 1 and 10 threads.

Tables 5.1, 5.2, 5.3 and 5.4 contain a summary of the performance measurement results, together with simulation results from both our previous and updated simulators. The columns with results from the performance experiments are marked 'Experiment'. The columns with simulation results are discussed in the next section.

Dispatcher threads	Receiver queuing time Experiment	Receiver queuing time Ex-tend PS	Receiver queuing time Ex-tend Fixed Prio	Receiver queuing time DAPS L24	Receiver queuing time DAPS PS	Dispatcher queuing time Experiment	Dispatcher queuing time Ex-tend PS	Dispatcher queuing time Ex-tend Fixed Prio	Dispatcher queuing time DAPS L24	Dispatcher queuing time DAPS PS
1	0.53	0.00	0.50	0.05	4.76	6.71	7.25	5.96	7.38	8.49
2	0.59	0.01	0.50	0.12	5.15	5.45	5.23	5.48	5.83	5.87
4	1.55	0.01	0.50	1.52	5.49	3.57	1.68	4.60	3.80	1.86
6	3.59	0.02	6.99	4.36	6.15	1.86	0.66	0.00	1.21	0.00
8	5.39	0.03	7.04	5.22	5.36	0.74	0.37	0.00	0.47	0.00
10	6.01	0.03	7.09	6.15	5.70	0.40	0.10	0.00	0.13	0.00

Table 5.1: Comparison of receiver and dispatcher thread queuing times

Dispatcher threads	Receiver holding time Experiment	Receiver holding time Ex-tend PS	Receiver holding time Ex-tend Fixed Prio	Receiver holding time DAPS L24	Receiver holding time DAPS PS	Dispatcher holding time Experiment	Dispatcher holding time Ex-tend PS	Dispatcher holding time Ex-tend Fixed Prio	Dispatcher holding time DAPS L24	Dispatcher holding time DAPS PS
1	0.92	1.69	2.33	0.97	1.72	6.30	6.36	6.42	6.25	6.39
2	0.95	2.10	2.34	1.04	2.20	8.86	10.32	10.94	8.44	11.79
4	1.73	2.52	2.34	2.27	2.73	10.14	14.62	17.66	10.07	17.01
6	3.64	2.69	6.72	4.90	2.90	8.65	16.45	5.75	5.91	18.59
8	5.66	2.74	6.72	6.00	2.81	6.84	16.95	5.75	6.79	18.22
10	6.19	2.73	6.72	6.41	2.93	6.36	17.16	5.75	5.97	18.83

Table 5.2: Comparison of receiver and dispatcher thread holding times

Dispatcher threads	Receiver 1st phase time Experiment	Receiver 1st phase time Extend PS	Receiver 1st phase time Extend Fixed Prio	Receiver 1st phase time DAPS L24	Receiver 1st phase time DAPS PS	Dispatcher 1st phase time Experiment	Dispatcher 1st phase time Extend PS	Dispatcher 1st phase time Extend Fixed Prio	Dispatcher 1st phase time DAPS L24	Dispatcher 1st phase time DAPS PS
1	0.91	N/A	2.28	0.93	1.63	6.28	N/A	5.77	6.17	6.35
2	0.91	N/A	2.29	0.93	2.07	8.82	N/A	6.03	8.42	11.71
4	1.11	N/A	2.29	0.99	2.58	10.08	N/A	6.95	10.15	16.90
6	1.16	N/A	0.93	1.02	2.73	8.59	N/A	5.71	8.10	18.47
8	1.14	N/A	0.93	0.99	2.64	6.76	N/A	5.71	6.73	18.10
10	1.13	N/A	0.93	0.93	2.76	6.33	N/A	5.71	5.67	18.71

Table 5.3: Comparison of receiver and dispatcher 1st phase completion times

Dispatcher threads	Server response time Experiment	Server response time Extend PS	Server response time Extend Fixed Prio	Server response time DAPS L24	Server response time DAPS PS
1	14.43	15.31	14.52	14.57	21.23
2	15.81	17.66	14.30	15.26	24.80
4	16.18	18.83	14.34	16.34	26.83
6	15.11	19.82	13.62	14.40	27.35
8	13.92	20.08	13.65	13.39	26.10
10	13.72	20.02	13.68	13.14	27.16

Table 5.4: Comparison of middleware server response times

The presented values are all averaged over the 15000 requests. The measured receiver queuing time represents the time that requests are queued before the receiver thread (somewhere between client and server-side receiver thread, e.g. in socket buffers). The measured dispatcher queuing time represents the time that requests are queued at the dispatcher thread-pool. The measured 1st phase time of the receiver and dispatcher threads equal the (average) time between the arrival at the thread and departure of a request from the thread. After the 1st phase the thread enters the 2nd phase, where the thread is still busy, but no longer actually processing the request. The measured holding time of receiver and dispatcher threads consists of the 1st and 2nd phase time.

The CPU consumption of the receiver thread is 0.925 milliseconds per request. A dispatcher thread consumes 5.6 milliseconds per request, including the 5 milliseconds of the method executed by the target object. The changes in the queuing behavior for the receiver thread and the dispatcher thread-pool, when the size of the thread-pool is increased from 1 to 10, are clearly visible in the experimental results. The queuing time before the receiver thread grows, while the queuing time before the dispatcher thread-pool goes down. Because the dispatcher thread's priority is often higher than the receiver thread priority for a higher number of dispatcher threads, the receiver thread often has to wait for a dispatcher thread to finish processing the request before it can finish the 2nd phase. This causes higher holding times for the receiver thread, and thus higher queuing times before the receiver thread.

The effects of changing thread priorities are also visible in the 1st phase completion times of the dispatcher threads. When looking at the experiment results for the 1st phase completion times, we first see a growth and then around 5 dispatcher threads the 1st phase completion times go down again. The growth is caused by a dispatcher thread being interrupted by another thread (receiver thread) in the system, because it has a higher priority. The dispatcher thread has to wait in runnable state for some time before it returns to running state again, causing higher completion times. The priorities of the dispatcher thread get higher as more dispatcher threads are added, causing less forced context switches of the dispatcher threads, and thus lower completion times.

5.3.3 Validation results

Tables 5.1, 5.2, 5.3 and 5.4 contain simulation results from both our previous and updated simulators, along with the experimental results discussed in

the previous section. The columns with values from the first performance model implementation in Extend, using the processor sharing scheduler, are marked 'Extend PS'. The columns with values of the new Extend performance model, using the fixed priority scheduler, are marked 'Extend Fixed PRIO'. The columns with values of the DAPS performance model using the Linux 2.4 scheduler abstraction are marked 'DAPS L24'. The columns with values of the DAPS performance model, using the processor sharing scheduler, are marked 'DAPS PS'.

Our first performance model in which we modeled the scheduling behavior using a processor sharing node yielded encouraging performance predictions, but overestimated the holding times of the dispatcher threads and underestimated receiver thread holding times and queuing times for the receiver threads. As shown in Table 5.2, the dispatcher thread holding times in the Extend simulation are much higher than measured during the experiments. These higher holding times are caused by the processor sharing scheduling, which effectively schedules all runnable threads with an infinitely small time quantum. The server response times were over estimated by the Extend simulation, shown in Table 5.4, mostly because of overestimation of the dispatcher thread holding times. The receiver holding times were underestimated, shown in Table 5.2, again, caused by scheduling behavior (receiver threads don't have to wait for a higher priority dispatcher thread to finish processing when using processor sharing scheduling). The underestimation of the receiver holding times in turn caused underestimation of the receiver queuing times, shown in Table 5.1.

After the first performance model in which we modeled the CPU resource by a queuing node with processor sharing (PS) discipline we replaced the PS node with a sub-model that more accurately captures the priority scheduling features described in Section 4.7. As shown in Table 5.1 and Figure 7-1, these two new performance models correctly predict the queuing behavior of the receiver and dispatcher threads. The breakpoint around 5 threads (see Section 4.7.2), where the dispatcher threads priorities are higher than the receiver thread priority, was correctly predicted by both new models. The abrupt changes in the queuing predictions made by the Extend fixed priority model are expected. In the real system the thread priorities are dynamic. However, in the Extend model the priorities of the receiver and dispatcher threads are fixed. In the DAPS L24 model the priorities are dynamic, like in the real system, resulting in a more accurate prediction of queuing times. Implementing a more elaborate scheduler in Extend is possible, but not trivial. Extend is a more general purpose simulator, while DAPS is designed from

ground-up to model software interactions, making it easier to implement schedulers.

Note: 1st phase completion times of the receiver and dispatcher threads were not available in the old Extend model, since that model didn't support the notion of 1st and 2nd phase processing.

Figure 5.2, Figure 5.3, Figure 5.4, and Figure 5.5 illustrate the queuing times, holding times, 1st phase completion times, and server response times of the experiments and various performance models.

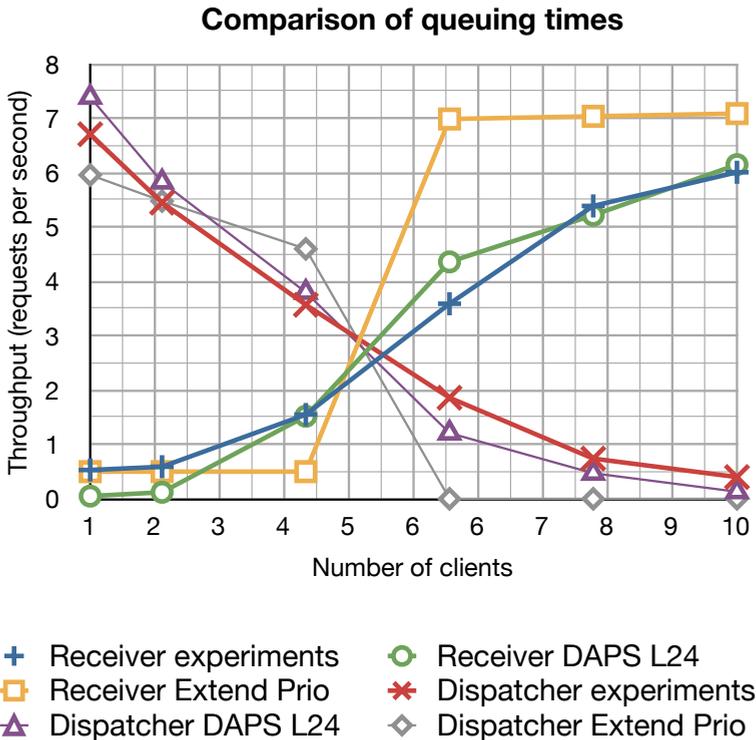


Figure 5.2: Comparison of queuing times

5.4 Validation of the threading strategies for an increasing number of clients

In the previous section we validated the performance model of the thread-pool strategy, simulated using Extend with a processor-sharing (PS) scheduler

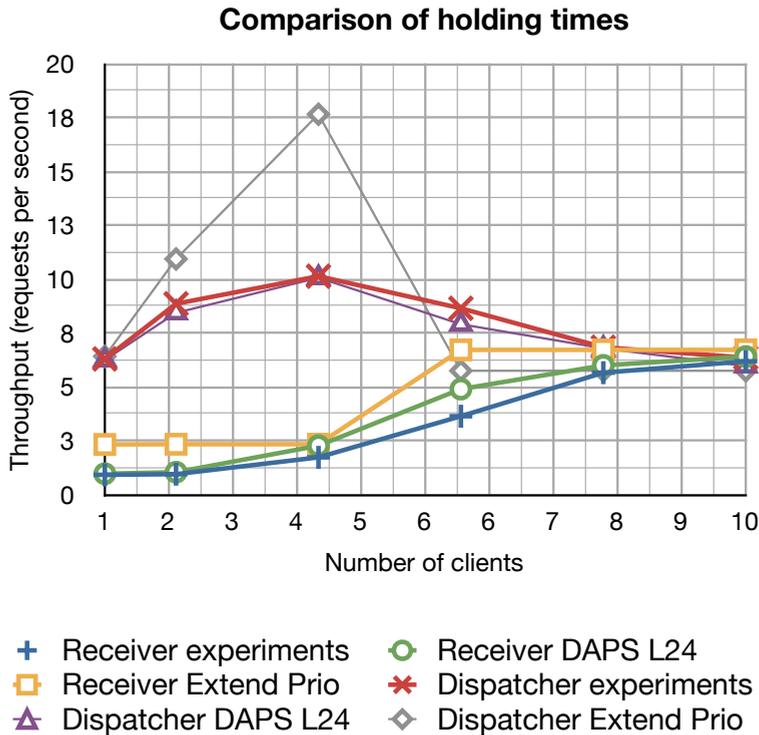


Figure 5.3: Comparison of holding times

and a fixed-priority scheduler, and using DAPS with a Linux 2.4 alike thread scheduling discipline. In this section we validate the models of the thread-per-request, thread-per-client, threaded and thread-pool strategies, using the DAPS simulation tool.

5.4.1 Experiment setup

In the experiments we use three different servant service demands. First, a 0.5 msec CPU demand, which represents the CPU processing cost of a simple method. Second, a 5 msec CPU demand, which represents the CPU processing costs of a more complex method (a CPU bound application). Third, a 50 msec delay (not CPU processing time), which represents the delay induced from a simple SQL query on a database server running on another machine (an I/O bound, database driven application). All service demands and delays are configured to have an exponential distribution.

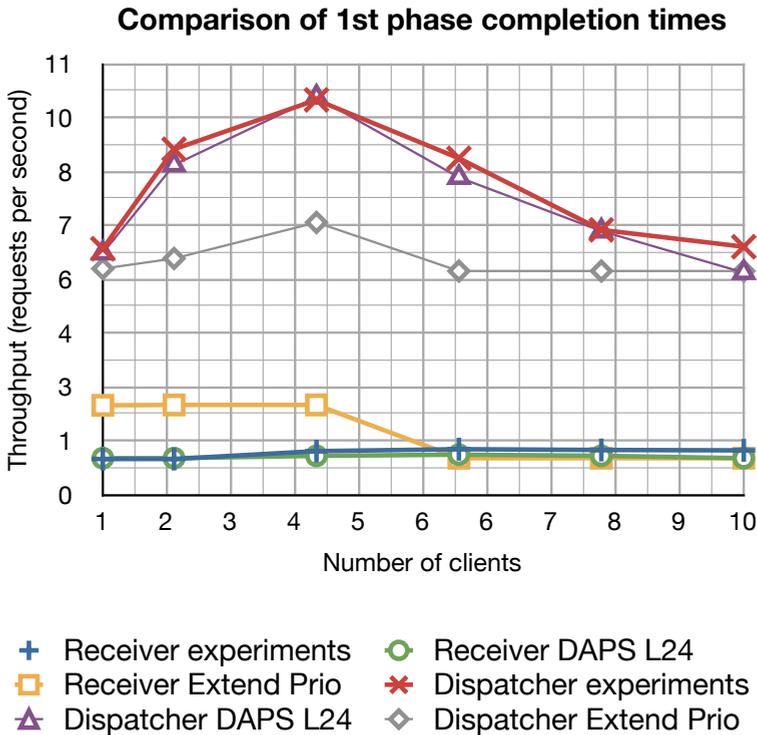


Figure 5.4: Comparison of 1st phase completion times

We compare the four threading models of ORBacus with an increasing number of connected clients. Each client has its own TCP/IP connection to the server ORB, and thus its own receiver thread on the server. Our test-bed consists of two machines: Utip267 and Utip442. Utip267 is a Pentium IV 1.7 GHz with 512 MB of memory. Utip442 is a Pentium III 550 MHz with 256 MB of memory. In these experiments Utip442 acts as the CORBA server and Utip267 as the CORBA client. Both machines run the Linux 2.4.19 operating system and the Sun Java 2 standard edition v1.4.1. The Java virtual machine is configured with default garbage collection settings and without run-time pre-compilation optimization features. The CORBA implementation we use in this example is IONA ORBacus/Java 4.1.1. We have configured our CORBA workload generator to send one request after another for each client (i.e. there is at most one active CORBA method invocation for each individual client). The thread-pools used in this experiment hold a number of threads equal to the number of clients.

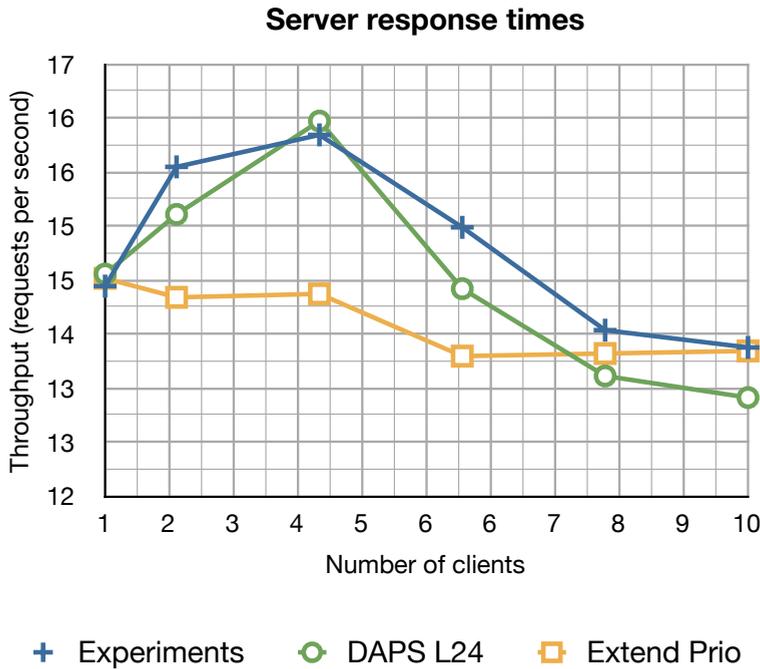


Figure 5.5: Server response times

In this experiments we use the following workloads as described above:

- 0.5 msec CPU demand
- 5 msec CPU demand
- 50 msec delay (not CPU demand)

We run the experiment with 1, 2, 4, 8, 16, 32 and 64 clients. Some experiments are also executed with 128 clients, depending on the CPU utilization at 64 clients. Each client executes a work-load of 200 requests on the server. We also have configured a minimum duration of 45 seconds for each experiment, so that we get enough measurements for runs with a small number of clients.

The following sections summarize the experimental and simulation results in tables for each experiment. Figures have been added to illustrate the experiment and simulation results.

5.4.2 Experiment results

This section summarizes the experiment results. Tables with detailed experiment results are included in the appendix. In all experiments the thread-per-client threading model is expected to be the most efficient, since we use single-threaded clients executing one blocking request at a time. In this scenario it doesn't make sense to release the receiver thread for processing forthcoming requests, since they won't arrive because the client is single threaded and blocking until it receives a reply for the current outstanding request.

0.5 msec servant CPU demand

Figure 5.6 contains the completion times for the different threading models. The thread-per-client and thread-pool threading models perform best in this experiment. Especially with a large number of clients, the ORB mutex present in the threaded threading model proves to be a bottleneck. Thread-per-request suffers from high thread creation and destruction costs, especially when compared to the small service demand of 0.5 msec CPU time. The thread-per-client and thread-pool threading models scale well with the number of connected clients. A knee can be seen for the threaded, around 64 clients. The overhead of thread-per-request is linear with the number of clients, but the completion times grow much faster than those of thread-per-client and thread-pool.

5 msec servant CPU demand

Figure 5.7 contains the completion times for the different threading models. Thread-per-client and thread-pool perform best of the four threading models. The contention for the ORB mutex of the threaded threading model causes it to be the worst performing threading model with a large number of clients.

The increasing performance difference between thread-per-client and the thread-pool may be surprising, but can be accounted to the Linux thread scheduler. The Linux thread scheduler favors threads that use small amounts of CPU time over threads that use larger amounts of CPU time. In this experiment we have increased the service demand of the servant from 0.5 msec to 5 msec, i.e. the CPU time required by the dispatcher thread has increased. The completion times for thread-per-client, thread-per-request, and thread-pool grow linearly with the number of clients. A curve can be seen for the threaded threading model.

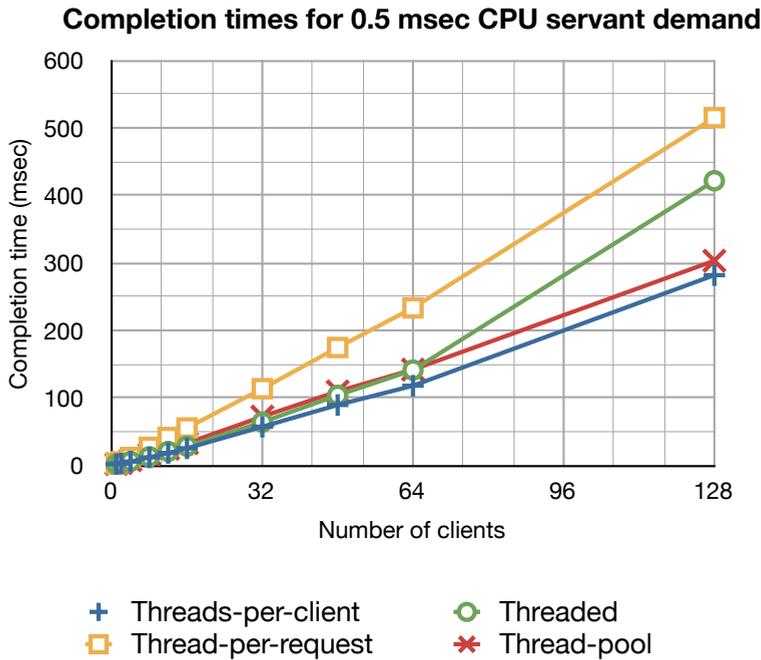


Figure 5.6: Completion times for 0.5 msec CPU servant demand

50 msec servant delay

Figure 5.8 contains the completion times for the thread-per-client, thread-per-request and thread-pool threading models. The performance of thread-per-client and thread-pool is almost exactly the same. Queuing delays in the thread-pool threading model remain low, because the servant execution (part of the dispatcher threads) only use a wall-clock delay (not doing any work on the CPU). The Linux thread scheduler does not penalize the dispatcher threads in this experiment. The completion time with the thread-per-request model grows a lot faster compared to thread-pool and thread-per-client. On a busy system the creation of new threads is delayed. The time between signaling the system for thread creation and the system actually creating and starting the thread is called ‘handover’ time in the experiment results. We can observe that the handover time grows with the number of clients, and accounts for a large portion of the completion time. We can also observe that the lowest measured completion time lies around 60 msec, where 50 msec could be expected. The difference can be explained by the granularity of the

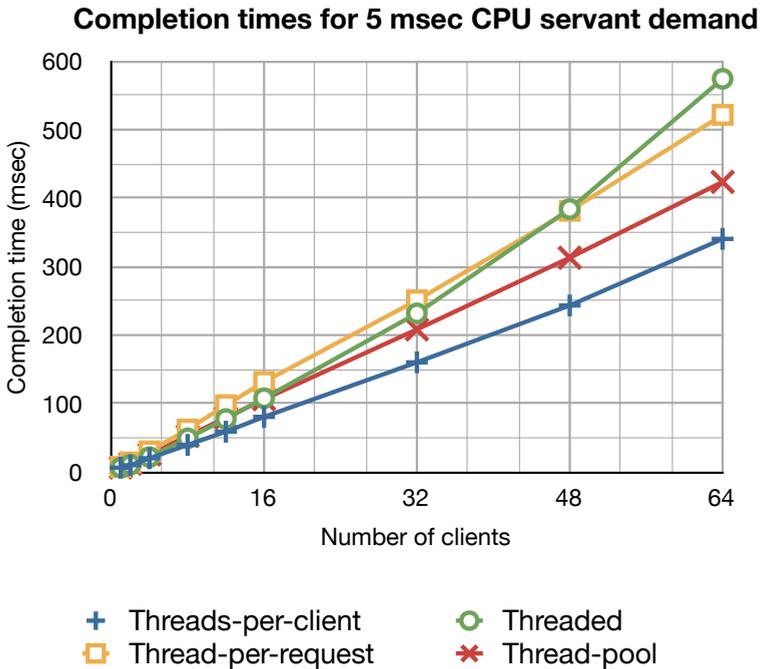


Figure 5.7: Completion times for 5 msec CPU servant demand

sleep() system call on Linux, which is 10 msec. When the sleep() system call is invoked, the thread has been active for a while already and already took a share of the current scheduler time-slice. Effectively, adding the 50 msec sleep interval would round the completion time up to 60 msec.

The completion times for the threaded threading model are plotted in Figure 5.9. The plot shows that the contention for the ORB mutex is responsible for the high completion times.

5.4.3 Validation results

We have configured the DAPS simulation tool to use the Linux 2.4 scheduler model, to schedule thread execution on the CPU resource in the model. In this section the experiment results from the previous section are compared with DAPS simulation results.

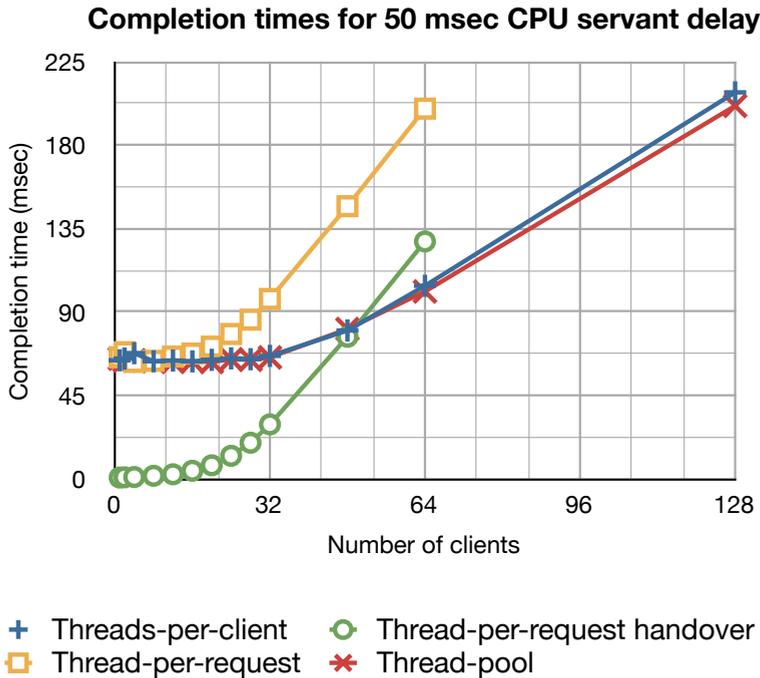


Figure 5.8: Completion times for 50 msec servant delay for the thread-per-client, thread-per-request and thread-pool threading models

Thread-per-client strategy

The thread-per-client threading strategy is the most basic threading strategy available. It uses a single thread to receive and process requests from the same client. This thread receives the request from the network, dispatches the request to the right POA and servant, executes the method invocation request on the servant, and sends back the reply to the client over the network.

Threaded strategy

The threaded threading strategy is a variation on the thread-per-client strategy. The only difference is that all requests are serialized by an ORB-wide mutex. Only one method invocation can be processed by any of the servants at a time. This threading strategy can be used for applications that are not multi-thread aware.

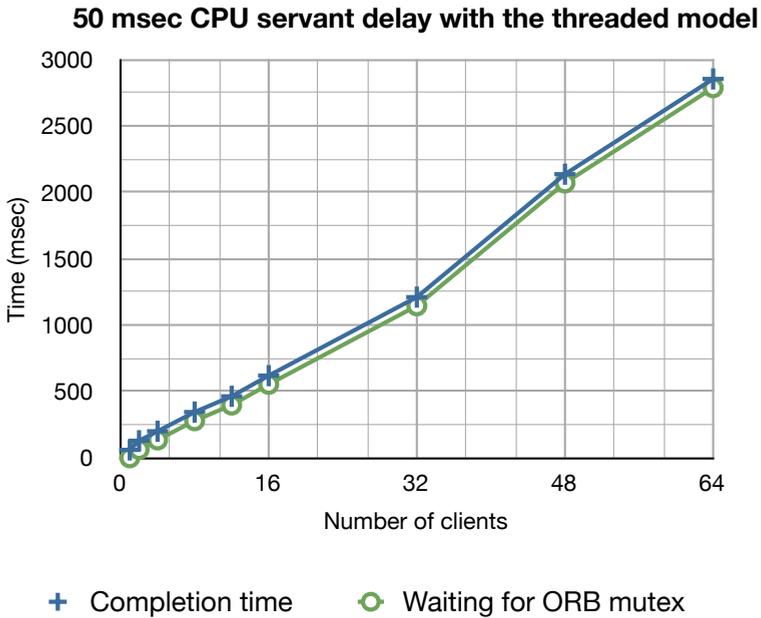


Figure 5.9: Completion times and ORB mutex contention for 50 msec servant delay with the threaded trading model

Clients	Experiment response time	Simulation response time
1	1.96	1.83
2	2.79	3.62
4	5.72	6.75
8	11.92	12.84
12	18.46	19.26
16	24.46	24.74
32	57.33	56.53
48	89.60	86.34
64	118.10	115.16

Table 5.5: Thread-per-client experimental and simulation results for 0.5 ms servant CPU demand

Thread-per-request strategy

The thread-per-request threading strategy separates the receiving and dispatching phases of requests into two separate threads. A receiver thread is allocated for each client (for the duration of the client-server binding). The receiver thread receives the request from the network, and locates the right POA to forward the request to. Then a dispatcher thread is allocated (one new thread for each request), and the request is forwarded to that thread for

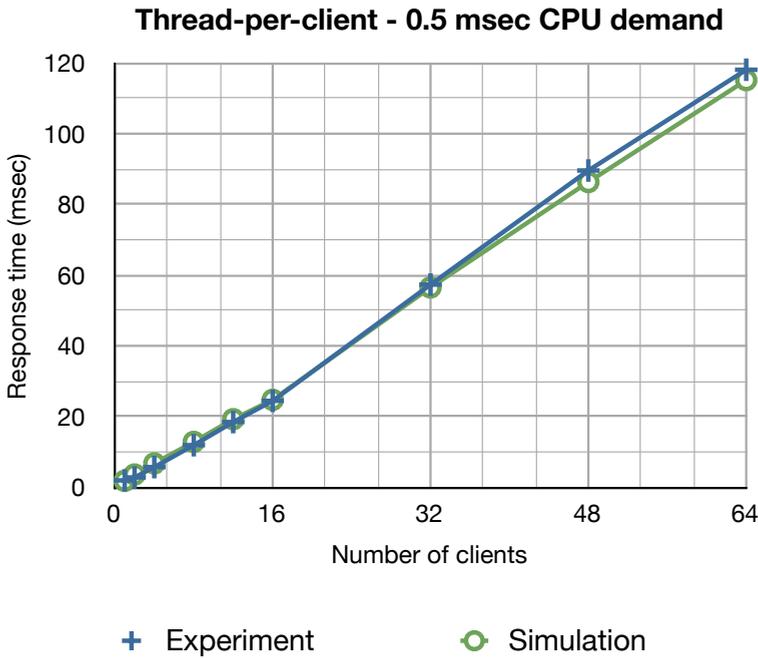


Figure 5.10: Thread-per-client response times for 0.5 ms servant CPU demand

Clients	Experiment response time	Simulation response time
1	6.50	5.94
2	10.00	11.50
4	20.17	22.33
8	39.70	43.61
12	59.01	66.16
16	80.61	87.46
32	160.24	177.84
48	243.33	273.72
64	340.52	360.77

Table 5.6: Thread-per-client experimental and simulation results for 5 ms servant CPU demand

the request dispatcher phase. During the dispatching phase the dispatcher thread dispatches the request to the right servant, where it is executed. After the method invocation on the servant, the dispatcher thread sends the reply to the client. After creating the dispatcher thread, the receiver thread can process new requests by the same client. So, at the same time multiple dispatcher threads may be processing requests for the same client. That is

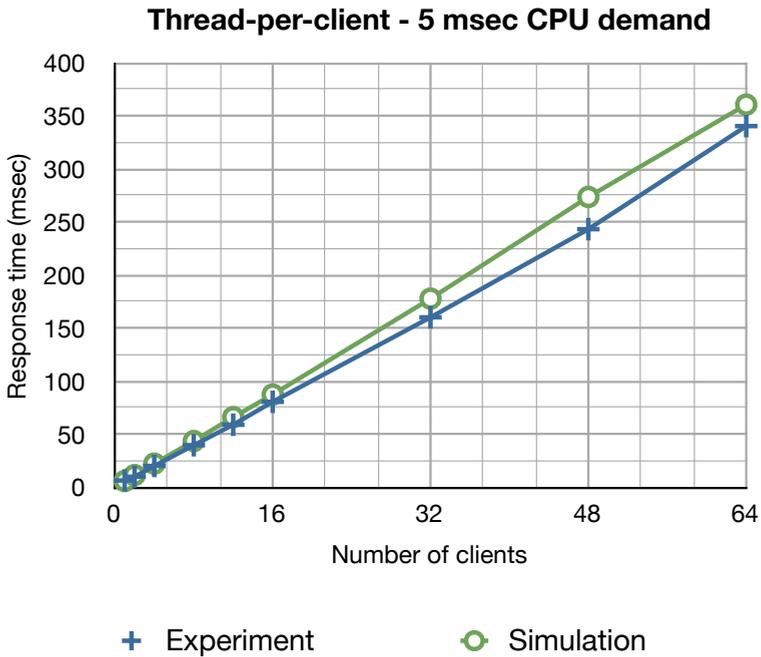


Figure 5.11: Thread-per-client response times for 5 ms servant CPU demand

Clients	Experiment response time	Simulation response time
1	64.00	64.92
2	64.93	64.87
4	67.83	64.86
8	63.46	64.77
12	63.87	64.67
16	63.42	64.70
32	66.38	65.57
48	80.07	82.05
64	104.21	105.78

Table 5.7: Thread-per-client experimental and simulation results for 50 ms servant delay

the advantage of this threading strategy; it allows multiple requests from the same client to be processed concurrently. The disadvantage of this threading strategy is that it creates a new thread for each request, and destroys the thread after executing the request.

The simulation results show that the ‘handover’ time (can also be thought of as ‘setup’ time for the thread) is under-estimated. The handover time measure represents the time between creation of the thread and the time the

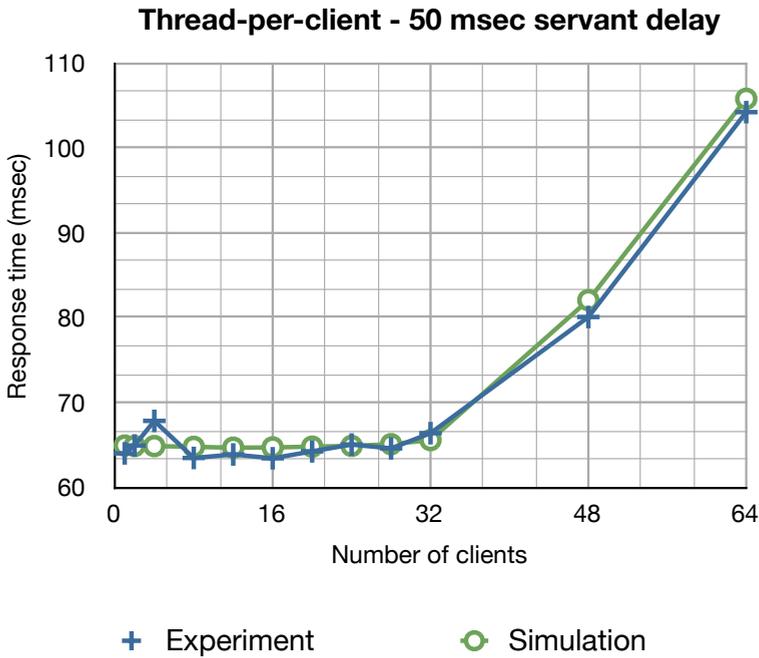


Figure 5.12: Thread-per-client response times for 50 ms servant delay

Clients	Experiment response time	Experiment mutex contention	Simulation response time	Simulation mutex contention
1	2.51	0.01	1.89	0.00
2	3.60	0.07	3.66	0.00
4	6.77	0.66	6.62	0.40
8	14.06	4.12	12.60	3.22
12	21.69	8.95	18.59	7.21
16	29.82	15.05	27.72	13.65
32	63.32	39.35	58.34	38.47
48	93.02	62.65	90.02	69.89
64	131.96	92.19	124.84	103.33

Table 5.8: Threaded experimental and simulation results for 0.5 ms servant CPU demand

thread actually starts to execute the request, i.e. when it gets scheduled by the operating system's thread scheduler. In our simulation model the new dispatcher thread can start to execute the request as soon as it gets scheduled by the scheduler. However, in reality there are a number of interactions between the thread spawning the new dispatcher thread and the dispatcher thread, to setup the new thread. During these interactions the new thread gets scheduled in a couple of times. After the setup phase of the new dis-

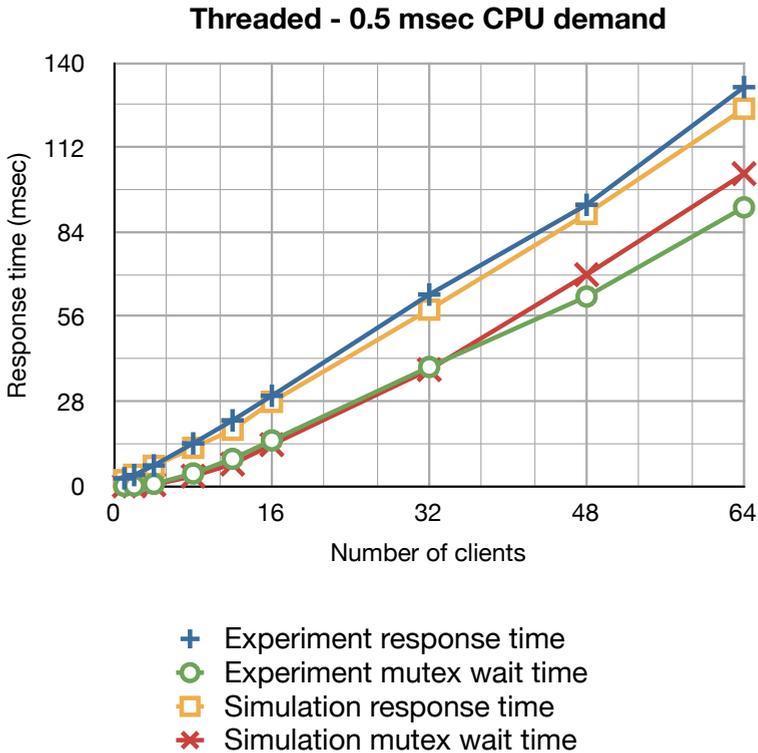


Figure 5.13: Threaded response times for 0.5 ms CPU servant demand

Clients	Experiment response time	Experiment mutex contention	Simulation response time	Simulation mutex contention
1	6.44	0.01	6.74	0.00
2	10.68	1.50	13.19	1.23
4	21.84	5.86	25.01	4.22
8	49.31	21.34	52.49	17.73
12	77.53	41.26	77.98	37.27
16	107.80	64.13	105.30	61.23
32	231.78	168.00	227.15	168.92
48	384.11	301.01	378.29	294.84
64	574.30	474.74	563.19	446.08

Table 5.9: Threaded experimental and simulation results for 5 ms servant CPU demand

patcher thread has completed, the receiver thread has to start the dispatcher thread (after setup the new thread is in suspended state). This behavior is really behavior of the Java platform (the VM in particular) rather than application or operating system behavior. We haven't captured this behavior in

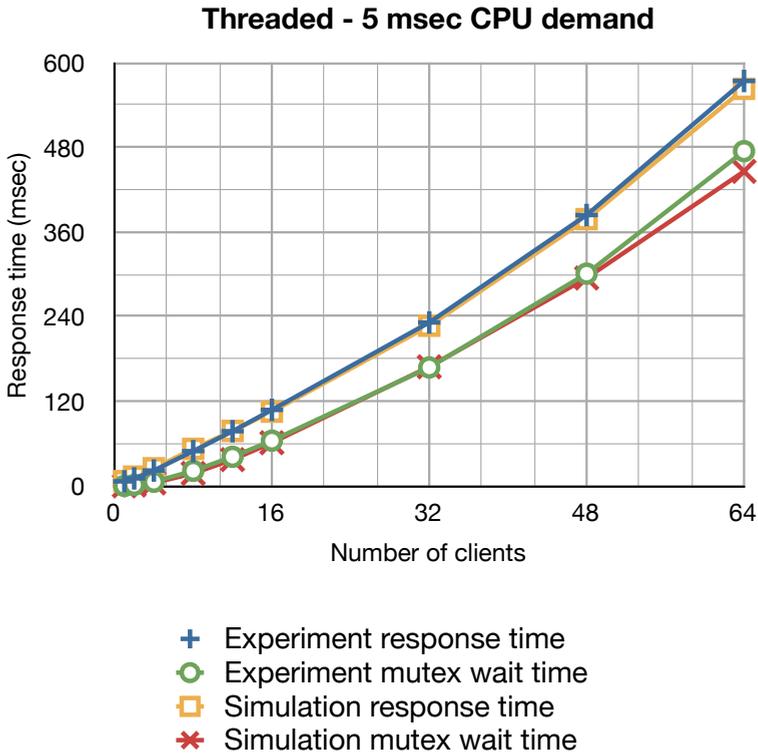


Figure 5.14: Threaded response times for 5 ms CPU servant demand

Clients	Experiment response time	re-	Experiment mutex contention	Simulation response time	re-	Simulation mutex contention
1	59.30		0.01	54.00		0.00
2	128.47		62.32	100.39		47.00
4	199.72		134.15	178.45		125.24
8	344.39		279.04	365.84		312.73
12	462.97		397.16	504.23		451.12
16	618.74		553.40	664.22		611.11
32	1210.12		1144.68	1379.29		1326.14
48	2136.09		2070.26	2092.14		2038.44
64	2853.70		2787.53	2849.30		2795.38

Table 5.10: Threaded experimental and simulation results for 50 ms servant CPU delay

our simulation model yet, therefore it underestimates the handover time.

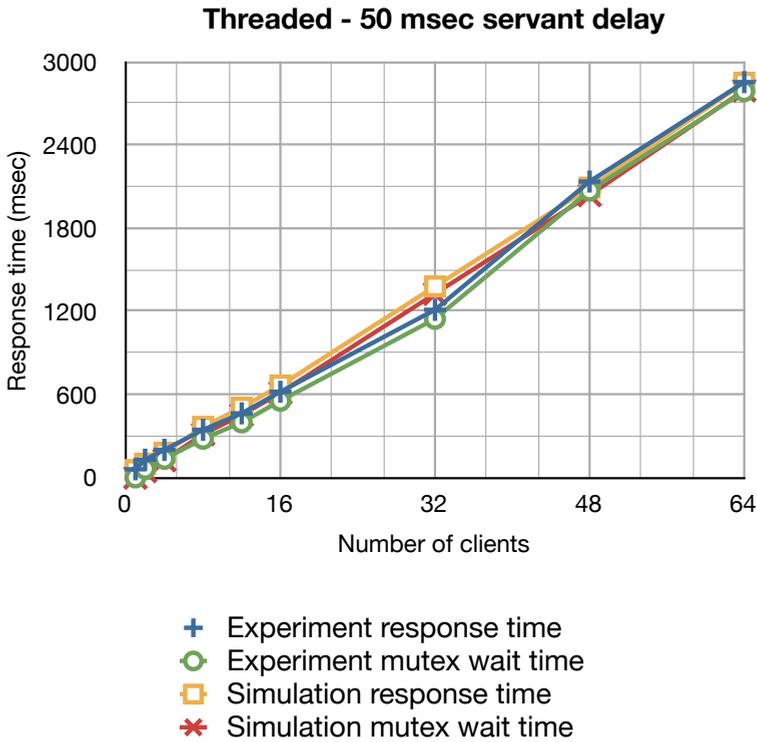


Figure 5.15: Threaded response times for 50 ms servant delay

Clients	Experiment response time	re-	Experiment han-	han-	Simulation response time	re-	Simulation han-	han-
1	2.78		0.68		2.98		0.00	
2	5.22		1.87		5.99		1.26	
4	11.85		7.15		14.15		3.35	
8	26.81		20.03		27.69		8.54	
12	41.34		33.66		41.54		13.24	
16	55.59		47.76		54.29		18.73	
32	113.71		104.64		112.19		38.92	
48	174.70		161.89		173.44		60.71	
64	233.46		219.57		231.04		81.32	

Table 5.11: Thread-per-request experimental and simulation results for 0.5 ms servant CPU demand

Thread-pool strategy

The thread-pool threading strategy is a refinement of the thread-per-request strategy. Instead of creating a new thread for every request, a pool of pre-allocated threads is used for request dispatching. Like the thread-per-request

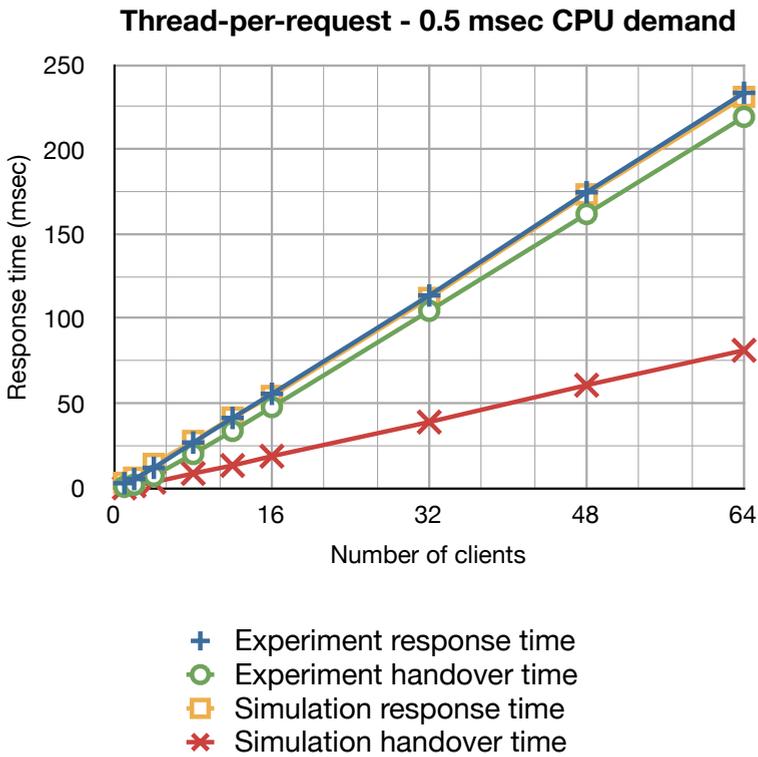


Figure 5.16: Thread-per-request response times for 0.5 ms CPU servant demand

Clients	Experiment response time	Experiment handover time	Simulation response time	Simulation handover time
1	7.48	0.71	8.09	0.00
2	14.16	3.80	15.97	1.53
4	30.25	13.60	34.26	8.85
8	62.24	42.18	67.56	20.84
12	97.72	70.99	102.40	33.09
16	131.67	104.07	134.34	44.98
32	250.90	216.93	269.99	91.73
48	381.50	348.97	402.34	140.43
64	521.64	483.89	532.88	184.34

Table 5.12: Thread-per-request experimental and simulation results for 5 ms servant CPU demand

strategy, a receiver thread receives requests from a client (one receiver thread per client). But instead of creating a new dispatcher thread, the receiver thread en-queues the request in a FIFO queue. This queue is processed

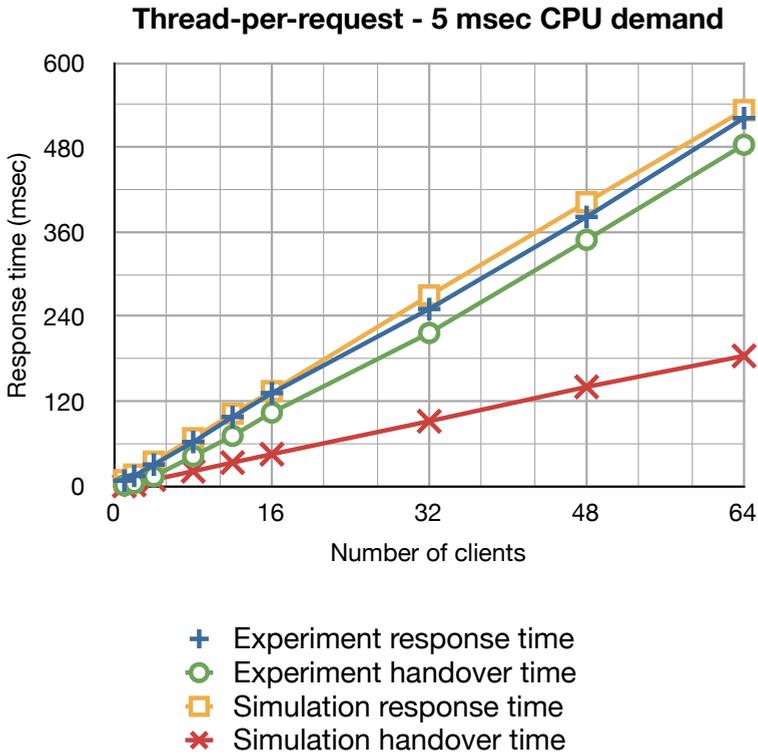


Figure 5.17: Thread-per-request response times for 5 ms CPU servant demand

Clients	Experiment response time	Experiment handover time	Simulation response time	Simulation handover time
1	65.63	0.73	64.71	0.00
2	68.39	0.84	64.01	0.55
4	62.13	1.00	64.12	0.84
8	63.73	1.67	65.07	1.07
12	66.03	2.48	65.16	1.22
16	67.63	4.38	65.67	1.62
32	97.10	29.41	67.92	3.26
48	147.18	76.68	91.13	18.79
64	199.84	128.13	125.29	40.42

Table 5.13: Thread-per-request experimental and simulation results for 50 ms servant CPU delay

by a fixed number of pre-allocated dispatcher threads (the thread-pool). Idle dispatcher threads obtain a request from the queue, and execute it. After execution and sending the reply to the client, the dispatcher thread is

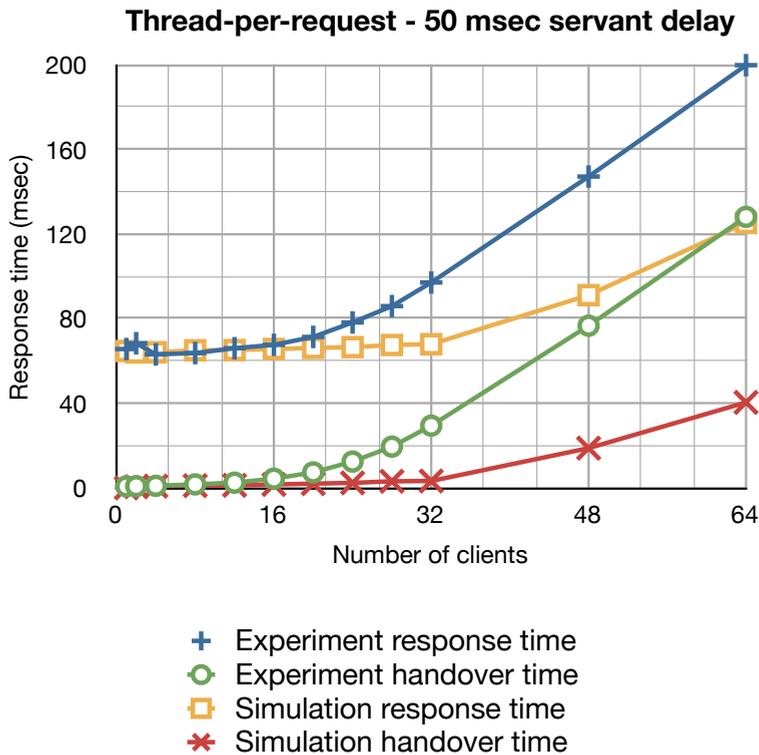


Figure 5.18: Thread-per-request response times for 50 ms servant delay

idle again and can process further requests. The advantage of this strategy compared to the thread-per-request strategy is that threads are pre-allocated, so thread creation and destruction does not occur for every request. Another advantage is that the thread-pool strategy can cope with bursts of requests. The number of simultaneously processing requests is bound by the size of the dispatcher thread-pool. With the thread-per-request strategy a request burst may cause a large number of threads to be created in a short amount of time, possibly causing stability problems. The disadvantage of the particular thread-pool strategy we've modeled (there are variations on this strategy, see Section 4.2) is that the threads are pre-allocated rather than created on demand. A dynamically growing and shrinking thread-pool can still have the advantages of the thread-pool studied here (not allocating and destroying threads for each request & a bounded number of threads), but doesn't need all threads pre-allocated.

Clients	Experiment re- sponse time	Experiment dis- patcher queuing time	Simulation re- sponse time	Simulation dis- patcher queuing time
1	2.13	0.09	2.39	0.00
2	3.49	0.55	4.79	0.94
4	7.77	0.85	8.68	1.76
8	16.37	1.15	16.63	2.40
12	24.65	1.59	24.73	2.21
16	32.94	1.80	33.46	2.60
32	72.41	4.11	74.10	6.08
48	109.70	5.65	112.31	8.61
64	141.93	7.48	148.59	10.14

Table 5.14: Thread-pool experimental and simulation results for 0.5 ms servant CPU demand

Clients	Experiment re- sponse time	Experiment dis- patcher queuing time	Simulation re- sponse time	Simulation dis- patcher queuing time
1	6.71	0.11	6.84	0.00
2	12.44	1.36	13.60	1.91
4	25.85	2.46	26.78	2.45
8	52.74	3.60	51.46	3.58
12	79.18	4.59	75.62	4.11
16	106.40	5.00	104.50	3.90
32	208.30	6.92	204.47	5.09
48	312.99	14.82	306.51	5.27
64	423.68	13.62	409.22	6.25

Table 5.15: Thread-pool experimental and simulation results for 5 ms servant CPU demand

Clients	Experiment re- sponse time	Experiment dis- patcher queuing time	Simulation re- sponse time	Simulation dis- patcher queuing time
1	64.92	0.04	64.71	0.00
2	63.97	0.06	63.78	0.01
4	63.99	0.08	63.78	0.01
8	63.50	0.08	64.56	0.06
12	63.16	0.09	64.40	0.15
16	63.49	0.10	64.45	0.22
32	65.56	0.32	65.72	0.60
48	80.89	0.97	84.93	2.86
64	101.15	1.96	108.22	2.39

Table 5.16: Thread-pool experimental and simulation results for 50 ms servant CPU delay

5.5 Summary

In this chapter we have discussed the simulation of the performance models and we compared the experimental results of various scenarios using four different threading strategies with simulation results. The simulation results match accurately with the experimental results, except for the ‘handover

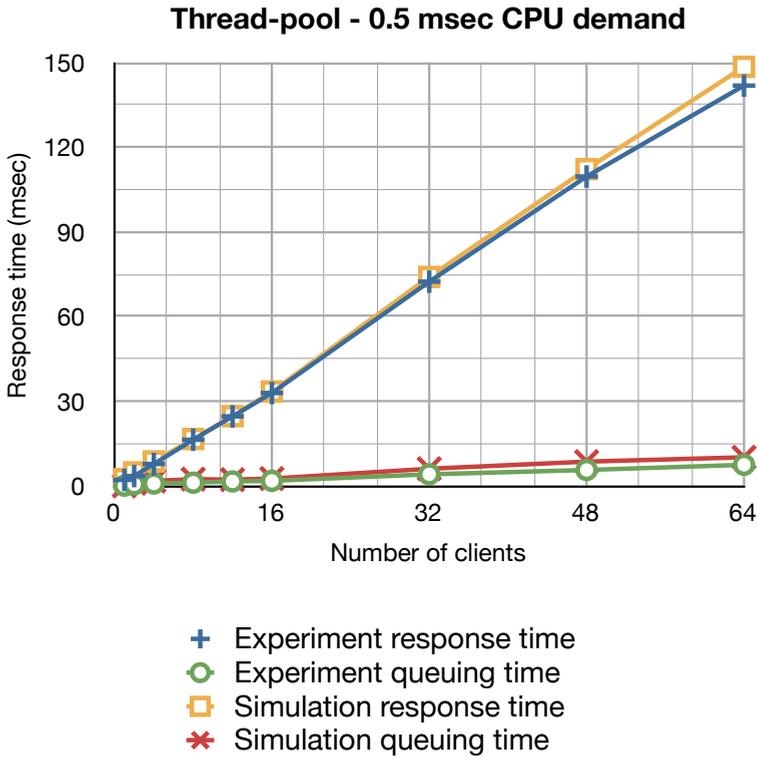


Figure 5.19: Thread-pool response times for 0.5 ms CPU servant demand

time' measure in the thread-per-request threading strategy. The handover times are under-estimated because our performance model does not capture some of the interactions in the Java virtual machine needed to setup new threads. Further study of this behavior is needed to refine our performance models.

In the next chapter we present a quantitative performance model for the end-to-end response time performance of a broad class of e-business applications, namely, interactive web-browsing (IWB) applications using middleware back-end servers.

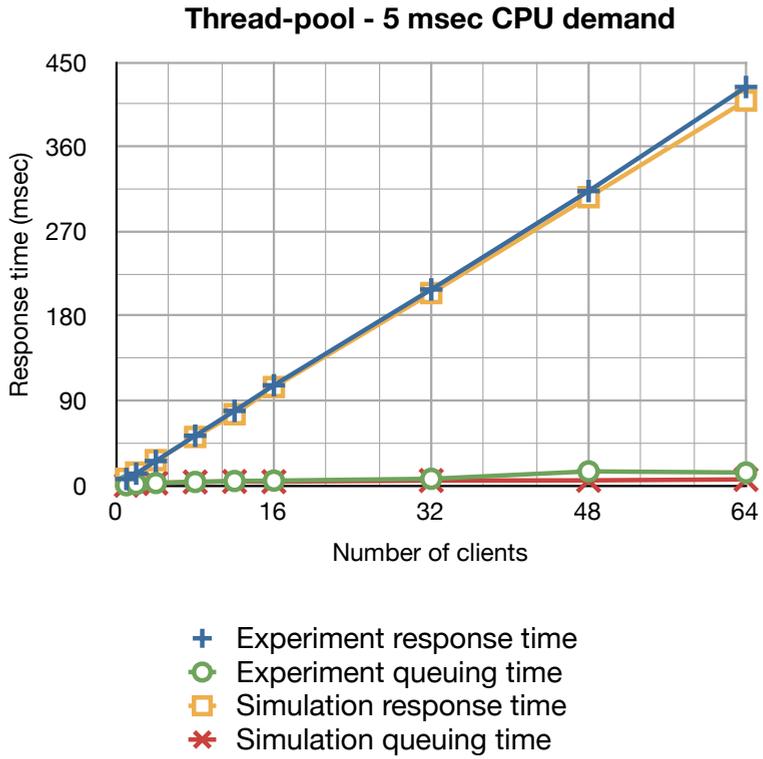


Figure 5.20: Thread-pool response times for 5 ms CPU servant demand

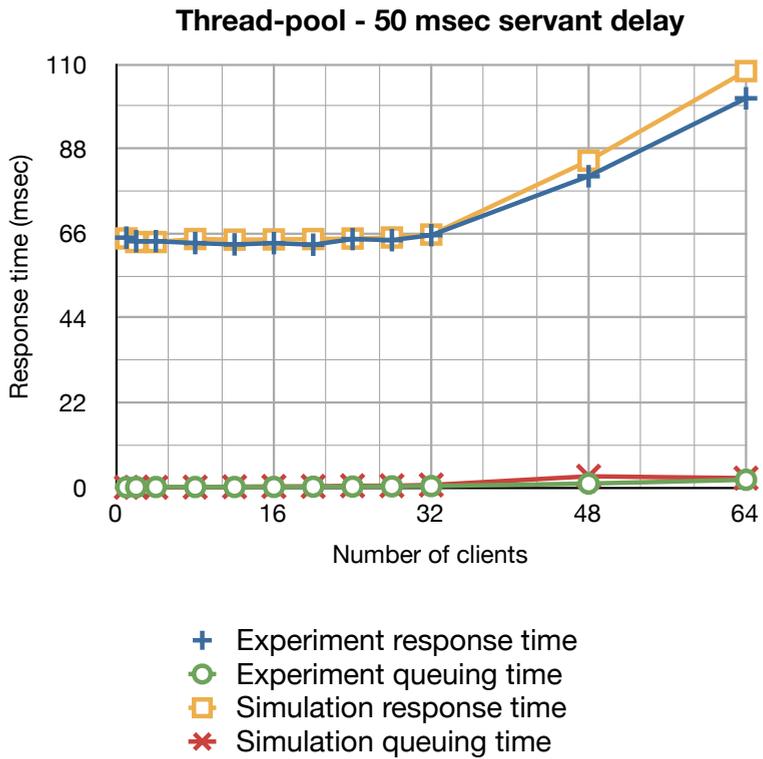


Figure 5.21: Thread-pool response times for 50 ms servant delay

PERFORMANCE MODELING OF AN INTERACTIVE WEB-BROWSING APPLICATION

In this chapter, we develop a quantitative performance model for the end-to-end response time performance of a broad class of e-business applications, namely, interactive web-browsing (IWB) applications using middleware back-end servers.

This chapter is structured as follows. Section 6.1 introduces our interactive web-browsing application use-case. Section 6.2 describes the local weather service application. Section 6.3 describes the performance model of the LWS. Section 6.4 describes the performance experiment results of the LWS. Section 6.5 presents simulation results and compares them with the experimental results. Section 6.6 summarizes this chapter.

6.1 Interactive web-browsing applications

An interactive web-browsing (IWB) application is a web based application that can dynamically create web pages depending on the user's input by combining and integrating information from different geographically distributed information systems. A key aspect that complicates the analysis of many e-business applications is their multi-domain nature: they com-

bine and integrate information from different geo-geographically distributed information systems, ranging over multiple administrative domains. In this context, the end-to-end performance experienced by the end user depends on many factors, such as combined performance of access and core networks, application servers, middleware, databases and operating systems. These observations raise the need for the development of performance models that describe the combined impact of these factors on the end-to-end user perceived performance. In this chapter we focus on the end-to-end response time performance, the key performance metric that determines the end-user's perception for interactive e-business applications [58].

To avoid being overly generic and less specific, we will present a model for our Local Weather Service (LWS), a specific but representative example of an IWB application that includes the main performance aspects of IWB applications (e.g., clients, web servers, middleware, operating systems, heterogeneity and multi-domain nature of the environment). The model is validated by comparing results from lab experiments with simulation results for a number of realistic workload scenarios. The results demonstrate that the performance predictions based on the model match well with the results from the lab experiments.

6.2 The local weather service application

In this section we describe the Local Weather Service (LWS). The LWS gives the user a weather forecast for the user's current location. The LWS runs on a distributed platform consisting of a mobile access network, an IP core network, a web server, a servlet (generating the local weather forecast webpage using various back-end servers), a location server (where locations of mobile phones can be looked up), a weather server (where local weather forecasts can be looked up), and various databases (where authentication, location, and weather data content is stored).

Figure 6.1 shows the interactions that take place for each local weather query. The following steps take place for each request:

- Step-1 The mobile phone user requests a local weather forecast from the web-server.
- Step-2 The web-server forwards the request to a servant. The servlet authenticates the client, followed by requests to the location and weather server, if the client authenticated, described in steps 3 and 4. After

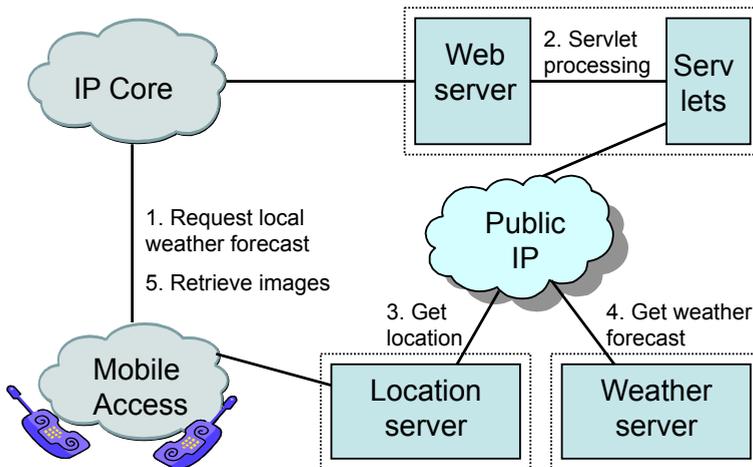


Figure 6.1: The local weather service

the requests to the location and weather server the servlet generates a dynamic web-page with the local weather forecast. This is forwarded to the web-server, which returns it to the mobile phone user. Weather images retrieved from the weather server are temporarily stored on disk, to be retrieved later by the user in subsequent requests to the web-server, see step 5.

Step-3 The servlet does a CORBA IIOP (object-oriented RPC) request on the location server, requesting the location of the mobile phone user.

Step-4 The servlet does another CORBA IIOP request, this time on the weather server, using the location of the mobile phone as a parameter. This request returns the local weather forecast, consisting of a text describing the weather, an image containing rain radar information, and temperature and weather type for the next three days.

Step-5 The web-browser of the mobile phone user receives the HTML page and sees a reference to the rain radar image. The browser issues another request to the web-server, this time a GET request, to obtain the image. The web-server receives the GET request and returns the image, stored on disk by the servlet, to the client.

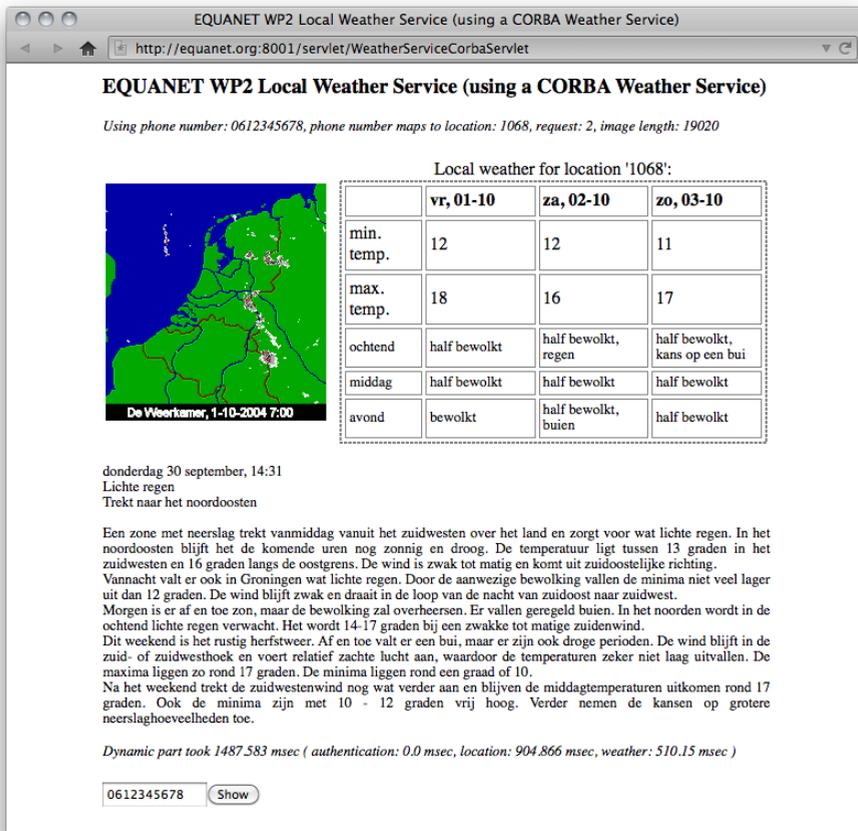


Figure 6.2: Screenshot of a web browser using the LWS

6.3 Performance model

In this section we present a model for the end-to-end response-time performance for the LWS. The model includes the combined impact of the session arrival process, web servers, middleware, databases, operating systems and communication networks. We emphasize that the end-to-end response-time performance generally depends on the specifics of these factors. Consequently, accurate performance models also depend on the specific choice of the web servers, middleware, databases, operating systems and communication networks. The model proposed below is based on the characteristics

of a W3C Jigsaw web server [62] with servlet technology, the ORBacus [31] CORBA object middleware [61] and the Linux operating system [6].

The system is composed of multiple sub-systems, each of which is dedicated to process a specific task. For example, in Figure 6.1 the sub-systems are (1) the web server responsible for handling the HTTP request and for creating dynamic web pages, (2) the location server responsible for retrieving the location information for the mobile phone user, (3) the location database, (4) the weather server responsible for retrieving the weather forecast, and (5) the weather forecast database. The retrieval of a weather forecast of a mobile phone user requires a number of interactions between these subsystems. Each subsystem follows a number of processing steps to execute its task. These processing steps require both physical and logical resources.

In the model for the web server the processing steps involved are the handling of incoming connections, the processing of requests arriving over these connections and the creation of dynamic web pages. These processing steps are implemented by an acceptor thread, an HTTP thread-pool and a servlet thread-pool, respectively. The HTTP thread-pool can dynamically grow towards a maximum number of threads (which can be configured). Idle threads are killed after a configurable amount of seconds. However, a minimum number of idle threads can be configured. The web server will not remove idle threads below that number. When connection requests arrive at the acceptor thread and the maximum number of client threads has been reached, the connection will be refused. The servlet thread-pool also dynamically grows, but no maximum is defined, or configurable. Idle servlet threads are removed after 24 hours (non configurable). However, the maximum is determined indirectly by the maximum number of client threads. We have investigated performance for stable systems, which is why we have not modeled the acceptor thread behavior of dropping requests when the maximum number of client threads has been reached. We also have not modeled thread creation and destruction costs, and thus also not thread idle times.

Both the location server and the weather server consist of application logic (servants) running on top of CORBA object middleware. In the models of these servers the processing steps are the receipt of the requests from the web server servlets, and the execution of the application logic. These steps are implemented by a receiver thread and a pool of dispatcher threads. The application logic can access database servers for information retrieval.

The middleware performance models are described in more detail in Chapter 4 of this thesis. Here we limit the model description of the middleware to

features relevant to the LWS application.

Note that the connections between the web server, and the location server and the weather server are established at application startup. Hence, connection setup processing does not occur on a per-request basis. For this reason, an addition acceptor thread is not included in the model of the middleware. Similarly, the connections between the middleware servers and the database servers also use persistent connections.

6.3.1 Queuing network

In this section we propose a queuing network to model the performance of the processing steps of the system (described above). To this end, the processing steps are modeled by nodes consisting of one or more servers representing logical resources (in our case: threads), and an infinite-size queue of pending requests that are served on a First Come First Served (FCFS) basis.

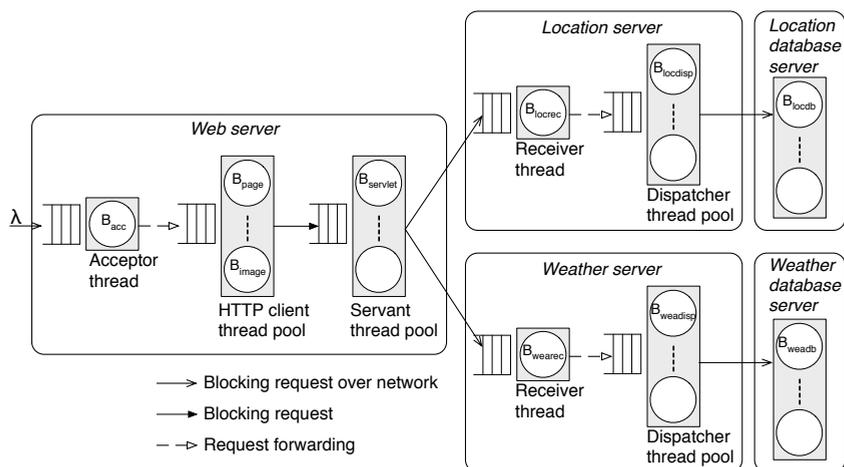


Figure 6.3: Performance model of the LWS application

Jobs arrive at the web server's acceptor node according to an arrival process with rate λ . (We emphasize we do not pose any restriction on the arrival process. In practice, one may use any synthetic or trace-driven workloads as the arrival process.) The service time at the acceptor thread is an independent random variable B_{acc} with a general distribution with mean β_{acc} . Jobs that find the acceptor thread busy upon arrival are placed in the queue.

After receiving service at the acceptor node the job is forwarded to the HTTP request processor node, which is equipped with C_{HTTP} threads. After forwarding the acceptor thread is free to process a new job. The service time of a job at the HTTP request processor node is a random variable B_{page} , with a general distribution with mean β_{page} . After receiving service at the HTTP node the job is forwarded to the servlet node, which is equipped with $C_{servlet}$ threads. The HTTP request processor node stays occupied (multiple resource possession) until the servlet node is done servicing the job. The service time of a job at the servlet node is a random variable $B_{servlet}$, with a general distribution with mean $\beta_{servlet}$. After receiving service at the servlet node the job is forwarded to the receiver node of the location server. The servlet node stays occupied until the location server is done servicing the job. At this point two threads in the web server are occupied for the job, a HTTP request processor thread and a servlet thread. The service time of a job at the receiver node of the location server is a random variable B_{locrec} , with a general distribution with mean β_{locrec} . After receiving service at the receiver node, the job is forwarded to the location request dispatching node, which is equipped with $C_{locdisp}$ threads. This node performs the location lookup application logic. The receiver node is free to process a new request. The service time of a job at the dispatcher node of the location server is a random variable $B_{locdisp}$, with a general distribution with mean $\beta_{locdisp}$. After receiving service the job is forwarded to the location database server. The dispatcher thread of the location server stays occupied. At this point there are two threads occupied in the web server and one thread in the location server. The database is modeled as an infinite server node. The service time of a job at the location database server is a random variable B_{locdb} , with a general distribution with mean β_{locdb} . After receiving service at the database node, the job is forwarded back to the dispatcher node at the location server. Then, the job is forwarded back to the servlet node at the web server. The dispatcher thread of the location server is released back into the thread-pool.

At this point the servlet knows the location of the mobile phone user and forwards the job to the receiver node of the weather server to obtain the weather forecast for that location. The servlet node stays occupied until the weather server is done servicing the job. At this point there are still two threads in the web server occupied for the job, a HTTP request processor thread and a servlet thread. The service time of a job at the receiver node of the weather server is a random variable B_{wearec} , with a general distribution with mean β_{wearec} . After receiving service at the receiver node, the job is

forwarded to the weather request dispatching node, which is equipped with C_{weadis} threads. This node performs the weather forecast application logic. The receiver node is free to process a new request. The service time of a job at the dispatcher node of the weather server is a random variable B_{weadis} , with a general distribution with mean β_{weadis} . After receiving service the job is forwarded to the weather database server. The dispatcher thread of the weather server stays occupied. At this point there are two threads occupied in the web server and one thread in the weather server. The weather database is also modeled as an infinite server node. The service time of a job at the weather database server is a random variable B_{weadb} , with a general distribution with mean β_{weadb} . After receiving service at the database node, the job is forwarded back to the dispatcher node at the weather server. Then, the job is forwarded back to the servlet node at the web server. The dispatcher thread of the weather server is released back into the thread-pool. The servlet is now done processing and forwards the job back to the HTTP processing node. The servlet thread is returned to the thread-pool.

At this point the web server has generated the dynamic web page with the weather forecast, and sent it to the client. However, the web page contains an embedded rain radar image which needs to be fetched from the web server. To fetch this image a new job is sent to the acceptor node of the web server. The service time and distribution for this job is the same as described above for the request of the web page, an independent random variable B_{acc} with a general distribution with mean β_{acc} . Jobs that find the acceptor thread busy upon arrival are placed in the queue. After receiving service at the acceptor node the job is forwarded to the HTTP request processor node, which is equipped with C_{HTTP} threads. After forwarding the acceptor thread is free to process a new job. The service time of a job at the HTTP request processor node is a random variable B_{image} , with a general distribution with mean β_{image} . After receiving service at the HTTP node, processing of the job is finished and the job departs the queuing network.

6.4 Experiments

In this section we describe our experiment setup and present the experiment results.

6.4.1 Setup

Our laboratory setup for the LWS consists of three machines. The HTTP load-generator runs on a Pentium III 800 MHz with 128 MB RAM. The web-server runs on a Pentium IV 1.7 GHz with 512 MB RAM. The middleware servers run on a Pentium III 550 MHz with 256 MB RAM. The HTTP load-generator has a 100 Mbit/s network connection with the web-server. The web-server has a 10 Mbit/s network connection to the middleware servers. Both middleware servers, the location server and weather server, run on the same machine.

All machines run the Linux 2.4 operating system and Java 2 standard edition v1.4.1. For this experiment we used high-resolution timers to generate accurate arrival processes. The CORBA middleware implementation we use is ORBacus 4.1.1 by IONA Technologies [31]. The web-server implementation we use is Jigsaw [62], a reference implementation web-server by the World Wide Web consortium (W3C).

Our LWS produces web pages around 2500 bytes, plus one image, around 25000 bytes. The 100 Mbit network from the load-generator to the web server has a RTT of 0.1 milliseconds. The 10 Mbit network of the web server to the middleware servers has a RTT of 0.5 milliseconds.

We have not used real database systems in our experiments, for practical reasons. The focus of our modeling work lies on application servers, web-server, middleware servers, and applications. In our experiments we have modeled the database accesses by delays, which can be configured by experiment parameters.

To obtain measurement data, we used our Java Performance Monitoring Toolkit (JPMT), described in Chapter 3 of this thesis. Amongst other measures, we monitored the following performance data during the experiments:

- Wall-clock completion times and CPU times of methods and threads in the load generator (client), web-server, and middleware servers
- CPU utilization, and
- Garbage collection.

We have performed two experiments. The first experiment, scenario A, studies how the LWS performs if there would be no delays (i.e. instantaneous database access by the authentication, location, and weather servers). This experiment stresses the CPU utilization of the servers to a point where one

of the servers becomes a bottleneck. The second experiment, scenario B, studies how the LWS performs if there would be delays induced by database accesses. In both experiments user session arrivals are described by Poisson processes. Paxson and Floyd have shown [43] [12] that Poisson distributions are valid for describing the arrivals of new user sessions. A session consists of two requests from the web-browser to the web-server. The first request initiates a new session, requesting a local weather forecast and yields a web-page, but without images. After receiving the web-page itself, the web-browser requests the images linked in the web-page from the web-server.

6.4.2 Scenario A

In the first experiment, scenario A, we configured all delays (authentication, location lookup time, and weather lookup time) to be 0.

Arrival rate (req/s)	Client-side total (ms)	Client-side page only (ms)	Client-side image only (ms)	Web server page (ms)	Web server image (ms)	Web server servlet (ms)	Location server (ms)	Weather server (ms)
0.98	93.78	55.90	35.88	56.21	33.91	54.74	3.67	32.68
4.57	106.66	63.50	42.00	61.83	39.69	58.46	5.10	33.36
8.56	130.93	75.72	54.06	65.62	49.41	63.34	7.54	34.56
10.28	146.49	80.79	64.90	73.69	67.02	68.68	9.20	35.58
15.20	241.25	125.88	111.04	105.09	104.37	93.53	16.04	43.89
16.61	284.55	162.69	119.25	114.39	112.36	114.53	21.12	48.78
17.52	328.54	197.83	127.24	121.33	112.71	131.20	25.11	54.31

Table 6.1: Measured wall-clock response times for scenario A

The response times we measured have been listed in Table 6.1. The 1st column specifies the measured rate of arrivals the load-generator generated. The 2nd, 3rd and 4th column show the response times at the load-generator (client) for the total reply (web page + image), the web page alone, and the image alone. The 5th and 6th column list the completion times of the client thread, for the web page and the image, respectively. The 7th column shows the completion time of the servlet thread, these are usually very close to the completion time of the client thread that handle the request to the servlet. The last two columns, the 8th and 9th, show the response times experienced by the web server for the calls to the location server and weather server, respectively.

Table 6.2 shows the CPU service demands of the client, the web server, the location server, and the weather server, needed for a single request. Note that the CPU times are dependent on the machines on which they are measured. It is clear that the web server is the bottleneck, even when it was running

Client-side total (ms)	Web server page (incl. Servlet) (ms)	Web server image (ms)	Location server (ms)	Weather server (ms)
3.65	21.07	29.00	1.95	4.06

Table 6.2: Measured CPU times for various parts of the request

on the fastest machine in our laboratory setup. At an arrival rate of 17.54 requests per second, the CPU utilization of the web server was around 88%, while the load-generator on the client, the location server, and the weather server all used less than 10% of the CPU. It is noteworthy that the CPU service demand required to handle the request to fetch the weather image is higher than the demand needed to create the HTML code of the web page of the weather forecast. The CPU service demand needed for handling the request to the weather image is used to calculate the MD5 checksum of the weather image.

Web server (ms)	Location server (ms)	Weather server (ms)
3	0	4

Table 6.3: Measured garbage collection times (averaged per request)

Table 6.3 shows the garbage collection times per request in the web server, location server, and the weather server. In practice they are not averaged per request. Instead some requests incur garbage collection during executions, while others do not. This introduces variance in the response times. Again, these times are machine dependent. These garbage collection times are not part of the CPU times we measure for each request. They are measured separately.

6.4.3 Scenario B

In scenario B we use delays to represent database query times for authentication, location lookup, and weather lookup. The configured delays, which are deterministic, are 30 milliseconds for authentication, 310 milliseconds for the location lookup, and 110 milliseconds for the weather forecast lookup.

Table 6.4 lists the measured wall-clock times for this scenario. The content of the columns is described in the previous section. In this experiment the CPU utilization of the web server was 12% at the highest. In this scenario the CPUs are not the bottleneck, but the number of threads allocated by the web server,

Arrival Rate (req/s)	Client-side total (ms)	Client-side page only (ms)	Client-side image only (ms)	Web server page (ms)	Web server image (ms)	Web server servlet (ms)	Location server (ms)	Weather server (ms)
0.52	547.25	513.17	34.10	513.35	35.74	508.95	313.93	143.00
0.72	547.70	512.48	35.22	509.72	34.98	508.72	313.93	142.64
1.03	550.88	514.16	36.72	512.45	36.50	509.90	314.07	143.39
1.18	552.88	515.17	37.70	513.51	39.10	509.80	314.17	142.98
1.52	554.72	517.49	37.23	515.59	36.94	510.67	314.44	143.23
1.76	557.93	517.95	39.98	514.88	40.15	510.85	314.38	142.93
2.04	559.40	518.09	37.79	517.28	35.97	511.96	314.88	144.22

Table 6.4: Measured wall-clock response times for scenario B

location, and weather servers are the bottleneck. Because of the large delays, the holding time for each thread is quite long compared to scenario A. When looking at the configured delays alone, the holding times for the client thread of the web server are at least 450 milliseconds. So when the load generator send more than 1000/450 requests per second the number of client threads will grow towards the configured maximum. When the maximum is reached, further connections to the web server will be refused.

The garbage collection times for this scenario are similar to the ones in scenario A.

6.5 Validation

In this section we present the simulation results for scenarios A and B (described in the previous section). To assess the validity of the performance model, we compare these simulation results with the experiment results for these scenarios.

6.5.1 Scenario A

Table 6.5 contains the simulation results for scenario A. Figure 6.4 contains plots for the experimental end-to-end response times and the end-to-end response times from the simulation.

The curves of the measured and simulated response times in Figure 6.4 are alike. At higher arrival rates, the response times of the simulation grow faster than the experimental response times. In future work these differences will be further investigated, so that the model can be refined. We expect one of the reasons to be our modeling of garbage collection. In the simulation we used the same values for garbage collection overhead per request for

Arrival Rate (req/s)	Client-side total (ms)	Client-side page only (ms)	Client-side image only (ms)	Web server page (ms)	Web server image (ms)	Web server servlet (ms)	Location server (ms)	Weather server (ms)
0.98	74.42	40.30	30.47	39.27	29.09	35.31	4.95	30.36
4.52	83.32	47.57	32.09	45.37	29.60	39.99	8.67	31.28
8.46	99.45	58.98	36.82	54.61	32.34	46.69	12.29	34.04
10.28	114.69	69.59	41.41	63.02	35.74	51.82	13.78	37.17
15.00	252.72	153.68	95.37	139.09	84.74	89.01	20.44	58.10
17.54	402.05	281.70	116.64	262.82	102.14	153.61	25.18	104.01

Table 6.5: Simulated wall-clock response times for scenario A

each simulated arrival rate. In practice however, garbage collection does not have to occur during every request, some request incur garbage collection overhead, while other requests do not. Also, garbage collection runs at a lower priority than normal Java threads. At higher CPU utilizations the lower priority threads will execute less often, yielding other garbage collection behavior and overhead.

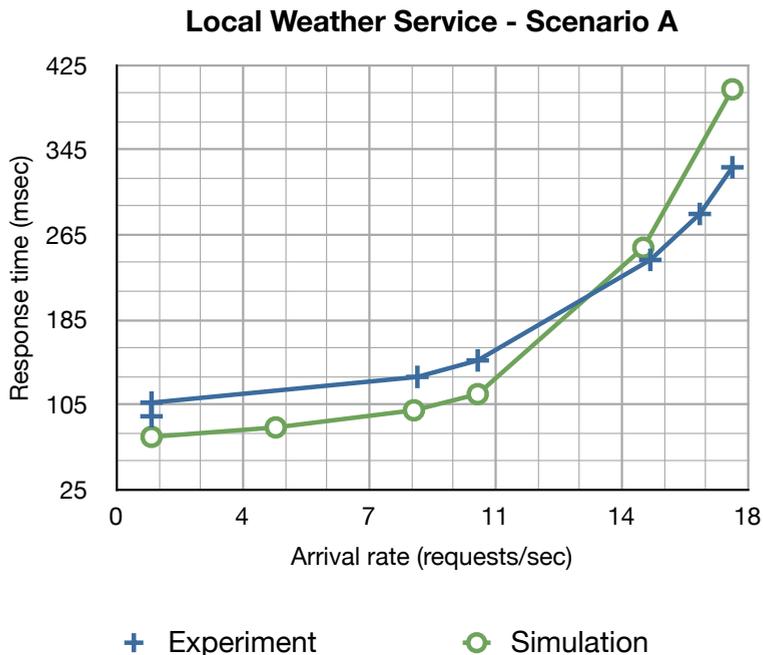


Figure 6.4: Experiment and simulation results for scenario A

6.5.2 Scenario B

Table 6.6 contains the simulation results for scenario B. Figure 6.5 contains plots for the experimental end-to-end response times and the end-to-end response times from the simulation.

Arrival Rate (req/s)	Client-side total (ms)	Client-side page only (ms)	Client-side image only (ms)	Web server page (ms)	Web server image (ms)	Web server servlet (ms)	Location server (ms)	Weather server (ms)
0.50	534.94	497.40	33.90	496.61	30.95	492.51	314.90	147.42
0.75	534.94	497.39	33.90	496.57	31.05	492.63	314.75	147.03
1.00	536.14	498.55	33.94	497.92	31.05	493.63	315.22	148.17
1.25	537.10	499.32	34.13	498.49	30.98	494.16	315.86	148.12
1.50	540.58	501.54	35.39	500.69	31.69	495.15	315.51	148.76
1.75	542.82	503.30	35.87	502.17	31.83	496.52	316.09	150.16
2.00	545.95	506.09	36.21	505.08	31.61	497.74	316.54	150.40

Table 6.6: Simulated wall-clock response times for scenario B

The curves of both measured and simulated response times, in Figure 6.5, are similar. However, there is a gap of about 10 milliseconds between the measured and simulated response times, a relative error of about 2 to 3%. We suspect these differences are caused by the granularity of the timer on the machines used in the experiments. Most operating systems (e.g. Windows and Linux) use a 10 millisecond timer resolution.

6.6 Summary

In this chapter we have reported on our performance experiments for our Local Weather Service, an interactive web-browsing application. We have developed a simulation model for this application, for which we have done initial validation using two different scenarios. The simulation model was able to predict the response times experienced by the client very well.

The simulation model can be refined in a number of directions. First, the impact of garbage collection on the performance was often found to be significant during the experiments. Therefore, the model may be extended to include the impact of garbage collection on the response-time performance. Second, the network delay model needs to be refined.

The experimental results show that the model accurately predicts the end-to-end performance for different load scenarios. In particular, it was found that the model accurately predicts the infamous “engineering knee”, i.e. the load values for which performance degradation becomes significant. This

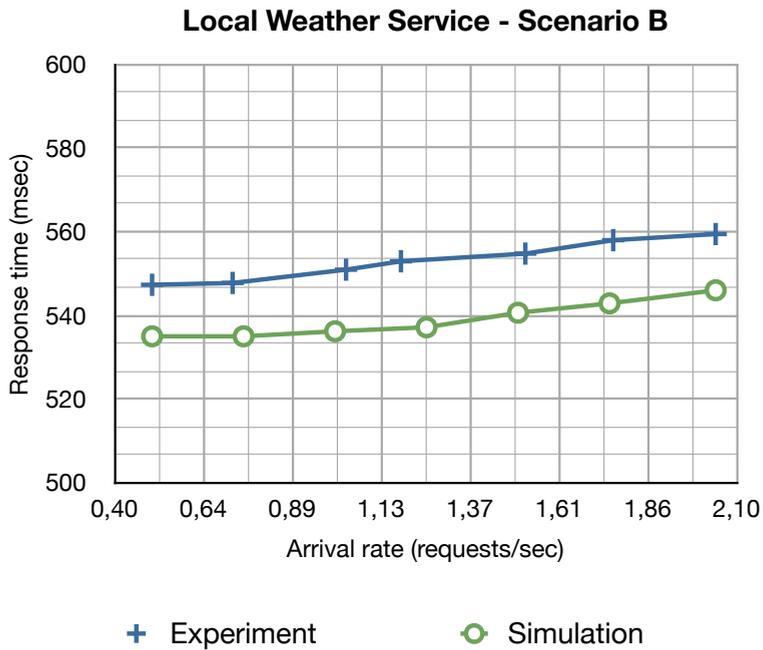


Figure 6.5: Experiment and simulation results for scenario B

observation opens the possibility for the implementation of effective Web Admission Control (WAC) schemes to prevent the system from performance degradation due to overload [14]. Analyzing the effectiveness of such a WAC scheme with our model-based predictions is a challenging topic for further research.

In the next chapter we will conclude this thesis.

CONCLUSIONS

In this chapter, we present the conclusions of this thesis and make suggestions for further research.

This chapter is structured as follows. Section 7.1 reviews the thesis objectives. Section 7.2 discusses further research options.

7.1 Review of the thesis objectives

This thesis has contributed in the development and validation of quantitative performance models of CORBA object middleware. In this chapter we review the thesis objectives.

The overall objective of this thesis was to develop and validate quantitative performance models of distributed applications based on middleware technology. In order to reach the overall objective, we have split this objective into the following sub-objectives:

1. Investigate and develop techniques to identify and quantify performance aspects of Java applications and components. These techniques will enable us to learn about performance aspects of software, and to quantify these performance aspects.
2. Obtain insight in the performance aspects of the Java virtual machine.

3. Obtain insight in the performance aspects of CORBA object middleware.
4. Obtain insight in the impact of multi-threading and the influence of the operating system's thread scheduler on the performance of threaded applications.
5. Combine these insights to construct quantitative performance models for CORBA object middleware.
6. Validate these performance models by comparing model-based results with real-world measurements.

In the following sections we come back to these objectives.

Develop insight in the performance aspects of CORBA, Java and threading

In Chapter 2 we have studied various techniques for measuring performance aspects in Java applications, the Java virtual machine, the operating system and the network. These techniques form the basis of our Java Performance Monitoring Tool (JPMT) that we have introduced in Chapter 3. Using this tool we can learn about the performance behavior of Java applications. The tool can be used to obtain insight in the execution behavior of an application. It does this by providing detailed execution traces of the application and its interaction with the underlying Java virtual machine and the operating system. The tool also provides measurements of this execution behavior. The tool combines monitoring results of instrumentation of the Java virtual machine, the system libraries and the operating system kernel into a complete picture of the application behavior.

Construction of performance models

Construction of performance models require insight in the performance behavior of an application. This insight can be obtained from studying documentation and source code, but also from studying the execution behavior of applications under various experimental workloads. The results of this study can be used to identify the parts of the application that are relevant to performance modeling. During this 'learning' phase, the application JPMT instruments the application very broadly, in order to obtain as much information as possible. A performance model can be constructed from what we have learned. To obtain input parameters for the performance model, JPMT can be used to only instrument the parts of the application that are relevant to the performance model, minimizing the overhead of the instru-

mentation. In Chapter 4 we construct performance models for a CORBA object middleware implementation.

Validation of these performance models

Chapter 6 discusses the implementation and the validation of the performance models. We developed a library for simulating performance aspects of distributed applications, called DAPS (Distributed Application Performance Simulator, Section 5.2), and implemented the performance models of Chapter 5 on top of this library. To validate these performance models we have compared simulation results with real-world experiment results for various workloads. The simulation results match accurately with the experimental results, except for the predicted time between the creation of a new dispatcher thread and the time this thread actually starts to execute the request in the thread-per-request threading strategy. This ‘handover time’ is probably under-estimated because our performance model does not capture some of the interactions in the Java virtual machine needed to setup new threads. This requires further research.

7.2 Future work

On the performance models

In the past chapters we identified several areas in which the the performance models could be improved. First, in Section 5.4.3 we have seen an inaccuracy in predicting the time between the creation of a new dispatcher thread and the time this thread actually starts to execute the request. The effect is discussed, but requires further study and validation. Second, in Section 4.6 we found the impact of Java garbage collection to be, at times, significant during our experiments. More research is needed on when to incorporate Java garbage collection in the performance models and how to do so. Third, the performance model could be extended to incorporate the aspects of CORBA marshaling, see Section 4.6. Fourth, the performance models could be extended by modeling the impact of the network in remote method invocations, see Section 4.3.

On the distributed application performance simulator

The DAPS simulation tool can be refined in several areas. First, the current implementation of the network ‘delay server’ is too simplistic for accurately modeling network response times for RPC and HTTP traffic. Second, the tool does not support multi-processor and multi-core machines. Third, the tool does not facilitate closed arrival processes. Fourth, resource constraints,

such as the maximum number of threads to allocate and maximum number of TCP connections, have not been implemented.

On the Java performance monitoring tool

The JPMT tool can also be improved in various areas. First, the current implementation only supports the JVMPI API. JVMPI, the Java Virtual Machine Profiling Interface, and JVMDI, the Java Virtual Machine Debugger Interface, have been deprecated in Java 5 and removed in Java 6. They have been replaced by the Java Virtual Machine Tool Interface (JVMTI) API. Second, the tool could be updated to use the Performance Application Programming Interface (PAPI), instead of accessing perfctr counters directly. This will make JPMT more portable (it currently only works on Linux). Third, JPMT produces event-traces for offline analysis. The tool could be extended towards supporting online monitoring scenarios and provide an API to obtain performance statistics of Java applications. This API can be used for controlling Quality of Service, monitoring service level agreements, and such.

REFERENCES

- [1] Ari, I., Hong, B., Miller, E., Brandt, S., and Long, D. Managing flash crowds on the internet. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)* (October 2003). 11
- [2] Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language*. Addison Wesley, 2000. 2
- [3] Balsamo, S., Marco, A., Inverardi, P., and Simeoni, M. Model-based performance prediction in software development: a survey. *IEEE Trans. on Software Engineering* 30, 5 (May 2004). 3
- [4] Beekhuis, R. Comparative performance evaluation of CORBA and DCOM. Master's thesis, KPN Research and University of Groningen, 1998. 81
- [5] Berrendorf, R., Ziegler, H., and Mohr, B. The performance counter library. <http://www.fz-juelich.de/zam/PCL>, 2003. 18
- [6] Bovet, D., and Cesati, M. *Understanding the Linux Kernel*, 2nd edition ed. O'Reilly Media, Inc., 2002. 85, 125
- [7] Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications* 14, 3 (2000), 189–204. 18, 45
- [8] Cantrill, B., Shapiro, M., and Leventhal, A. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference* (June 2004). 20
- [9] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. The state of art in locally distributed web-server systems. *CM Computing Surveys (CSUR)* 34, 2 (2002). 4

-
- [10] Chen, T., and Hu, L. Internet performance monitoring. *Proc. of IEEE 90*, 9 (September 2002), 1592–1603. 11
- [11] Corporation, I. *Intel Architecture Software Developer's Manual – Volume 3: System Programming Guide*. Intel Corporation, 1999. 46
- [12] Floyd, S., and Paxson, V. Difficulties in simulating the internet. In *IEEE/ACM Transactions on Networking* (August 2001), vol. 9, pp. 392–403. 130
- [13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995. 47, 55
- [14] Gijzen, B., Meulenhoff, P., Blom, M., van der Mei, R., and van der Waaij, B. Web admission control: improving performance of web-based services. In *Proc. of the 30th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG 2004)* (Las Vegas, Nevada, USA, 2004). 135
- [15] Gokhale, A., and Schmidt, D. Principles for optimizing corba internet inter-orb protocol performance. In *Proc. of HICSS '98* (Maui, Hawaii, 1998). 81
- [16] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification*, 3rd edition ed. Addison Wesley, 2005. 21
- [17] Grundy, J., and Hosking, J. *Wiley Encyclopedia of Software Engineering*, 2nd ed. Wiley Interscience, J. Wiley and Sons, 2001, ch. Software Tools. 4
- [18] Harkema, M. Instrumentation for performance measurements. Deliverable 2.1.1, Senter EQUANET, End-to-End Quality of Service in Next-Generation Networks, March 2003. 6
- [19] Harkema, M. Performance modeling of the equanet interactive web-browsing use-case. Deliverable 2.1.5, Senter EQUANET, End-to-End Quality of Service in Next-Generation Networks, November 2004. 6
- [20] Harkema, M. Performance model validation of middleware threading models. Deliverable 2.1.6, Senter EQUANET, End-to-End Quality of Service in Next-Generation Networks, February 2005. 6

- [21] Harkema, M., Gijsen, B., and van der Mei, R. Performance of middleware based architectures: A quantitative approach. In *30th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG 2004)* (Las Vegas, Nevada, USA, December 2004). 6
- [22] Harkema, M., Gijsen, B., and van der Mei, R. Report on validation of quantitative performance models of middleware. Deliverable 2.1.4, Senter EQUANET, End-to-End Quality of Service in Next-Generation Networks, February 2005. 6
- [23] Harkema, M., Gijsen, B., van der Mei, R., and Hoekstra, Y. Middleware performance: A quantitative approach. In *2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)* (San Jose, California, USA, July 2004). 6
- [24] Harkema, M., Gijsen, B., van der Mei, R., and Nieuwenhuis, L. Performance comparison of middleware threading strategies. In *2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)* (San Jose, California, USA, July 2004). 6
- [25] Harkema, M., Quartel, D., Gijsen, B., and van der Mei, R. Performance monitoring of java applications. In *ACM 3rd International Workshop on Software and Performance (WOSP 2002)* (Rome, Italy, July 2002), pp. 114–127. 6
- [26] Harkema, M., van der Mei, R., and Gijsen, B. Performance evaluation of middleware threading models and marshaling. Deliverable 2.1.3, Senter EQUANET, End-to-End Quality of Service in Next-Generation Networks, November 2003. 6
- [27] Harkema, M., van der Mei, R., and Gijsen, B. Report on basic quantitative performance models for middleware. Deliverable 2.1.2, Senter EQUANET, End-to-End Quality of Service in Next-Generation Networks, May 2003. 6
- [28] Harkema, M., van der Mei, R., Gijsen, B., and Quartel, D. Jpmt: a java performance monitoring tool. In *13th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2003)* (Urbana, Illinois, USA, September 2003). 6

- [29] Henning, M., and Vinoski, S. *Advanced CORBA Programming with C++*. Addison Wesley, 1999. 56
- [30] Hoare, C. Monitors: An operating system structuring concept. *Communications of the ACM* 17, 10 (October 1974), 549–557. 20, 32
- [31] IONA Technologies, Object Oriented Concepts Inc. ORBacus 4 for Java. <http://web.progress.com/en/orbacus/>, 2003. 125, 129
- [32] Jain, R. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, 1991. 3, 12, 13, 51, 61
- [33] Krahl, D. The extend simulation environment. In *Proc. of the 2000 Winter Simulation Conference* (Orlando, FL, USA, 2000). 87, 89
- [34] Lamport, L. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM* 21, 7 (July 1978), 558–565. 14
- [35] Lawrence Berkeley National Laboratory. Packet capture library (pcap). <http://www.tcpdump.org>, 2005. 30
- [36] Lazowska, E., Zahorjan, J., Graham, G., and Sevcik, K. *Quantitative System Performance*. Prentice-Hall Inc., 1984. 3, 61
- [37] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, 2nd edition ed. Sun Microsystems, 1999. 21, 47
- [38] M. Dahm, Date-Added = 2011-06-20 15:38:03 +0200, D.-M. . . I. . F M. . A. N. . B. T. . B. T. . T. Y. . . Tech. rep. 47
- [39] Menasce, D., and Almeida, V. *Capacity Planning for Web Services*. Prentice Hall, Inc., 2002. 14
- [40] Microsoft Corporation. Platform sdk: Performance monitoring. 2005. 20
- [41] Object Management Group. The Common Object Request Broker Architecture and Specification, revision 2.5. OMG document formal/2001-09-01, 2001. 1, 53, 56
- [42] Odlyzko, A. Internet traffic growth: Sources and implications. In *Proc. of Optical Transmission Systems and Equipment for WDM Networking II* (2003), B. Dingel, W. Weiershausen, A. Dutta, and K. Sato, Eds., no. 5247, The International Society of Optical Engineering (SPIE), pp. 1–15. 1

- [43] Paxson, V., and Floyd, S. Wide-area traffic: The failure of poisson modeling. In *IEEE/ACM Transactions on Networking* (June 1995), vol. 3, pp. 226–244. 130
- [44] Pettersson, M. The perfctr package for linux. <http://user.it.uu.se/mikpe/linux/perfctr/>, 2004. 45
- [45] Rolia, J., and Sevcik, K. The method of layers. *IEEE Transactions on Software Engineering* 21 (1995), 689–699. 3
- [46] Russinovich, M. Inside the Windows NT scheduler. *Windows IT Pro Magazine* (July 1997). 86
- [47] Schmidt, D., Stal, M., Rohnert, H., and Buscman, F. *Pattern-oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley Sons, 2000. 59
- [48] Schroeder, B. On-line monitoring: A tutorial. *IEEE Computer* 28, 6 (June 1995), 72–77. 11, 13, 14
- [49] Snodgrass, R. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems* 6, 2 (May 1988), 157–196. 13
- [50] Sun Microsystems. The java hotspot virtual machine. Technical White Paper, 2001. 2, 26
- [51] Svodobova, L. Performance monitoring in computer systems: A structured approach. *Operating Systems Review* 15, 3 (1981), 39–50. 11
- [52] Szyperski, C. *Component Software – Beyond Object-Oriented Programming*. Addison Wesley, 1999. 4
- [53] The Economist, and IBM Corporation. The 2004 e-readiness rankings. <http://www.eiu.com/>, 2004. 3
- [54] The Object Management Group. <http://www.omg.org/>. 1
- [55] Tran, P., Gosper, J., and Gorton, I. Evaluating the sustained performance of cots-based messaging systems. vol. 13. 2003, pp. 229–240. 11
- [56] Trivedi, K. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, 2nd ed. John Wiley Sons, 2002. 3
- [57] Vahalia, U. *Unix Internals – The New Frontiers*. Prentice Hall, 1996. 40, 86

- [58] van der Gaast, S., Beerends, J., Ahmed, K., and Meeuwissen, H. Quantification and prediction of end-user perceived web-browsing quality. *White contribution COM 12-C3 to ITU-T Study Group 12* (November 2004). 122
- [59] van der Mei, R., Gijzen, B., and van den Berg, J. End-to-end quality of service modeling of distributed applications: the need for a multidisciplinary approach. *CMG Journal on Computer Management* 109 (2003), 51–55. 3
- [60] van der Mei, R., and Harkema, M. Modelling end-to-end performance for transaction-based services in a distributed computing environment. In *Proceedings 1st Korea-Netherlands Joint Conference on Queueing Theory and its Applications to Telecommunication Systems (Seoul, June 2005)* (Seoul, South Korea, June 2005). 6
- [61] Vinoski, S. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine* 35, 2 (1997), 46–55. 1, 125
- [62] W3C. Jigsaw: W3C's Server. <http://www.w3.org/Jigsaw/>. 125, 129
- [63] Wilson, P. *Uniprocessor Garbage Collection Techniques*. LNCS 637. Springer Verlag, 1992, pp. 1–42. 21
- [64] Wisniewski, R., and Rosenburg, B. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. *Proc. of ACM Supercomputing 2003* (November 2003). 11
- [65] Woodside, C. Software performance evaluation by models. *Performance Evaluation, LNCS 1769* (2000), 283–304. 3, 31, 51
- [66] Woodside, C., Neilson, J., Petriu, D., and Majumdar, S. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers* 44, 1 (1995), 20–34. 3
- [67] Yaghmour, K., and Dagenais, M. R. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference* (June 2000). 20, 35, 36