University of
Department of Computer Science
Building 48, Room 375
Erwin-Schrödinger-Str.
67663 Kaiserslautern

Diploma Thesis

Design and Implementation of a Xen-Based Execution Environment*

Alexander Petry†

April 2007

Advisor: Juniorprof. Peter Merz

Co-Advisor: Dipl.-Wirtsch.-Ing. (Inf.) Mathias Dalheimer

* XenBEE: see `http://xenbee.berlios.de`

† registration number: 345466, e-mail: `<petry@itwm.fhg.de>`

Virtualization is a concept that one cannot think away from computer science anymore.

Preface

In the last few years the remote execution of applications has gained more and more on importance. This is due to the fact that paradigms like grid computing have been developed. The computation power that is required by a scientist to execute a complex simulation need not to be locally available anymore. Instead he can submit the simulation to a remote high performance cluster that is provided by an organization that has specialized in providing computation power.

The current developments in the area of hardware virtualization show that current desktop computer systems are powerful enough to execute many different operating systems in parallel. The usage of hardware virtualization imposes only a little overhead.

This work aims at sticking both concepts together to provide a novel execution environment. This environment is going to provide secure remote execution of applications in user-supplied virtual machines. The execution will behave like a batch job — send the execution and input data away and get the results back.

In den letzten Jahren hat die Ausführung von Applikationen auf entfernten Systemen mehr und mehr an Bedeutung gewonnen. Das liegt daran, dass Paradigmen wie zum Beispiel das Grid Computing entwickelt wurden. Die Rechenleistung, die ein Wissenschaftler für eine aufwändige Simulation benötigt, muss nicht mehr lokal zur Verfügung stehen. Vielmehr kann der Wissenschaftler seine Berechnung auf einem High-Performance Cluster durchfüren lassen, der von einer Organisation bereitgestellt wird, die auf die Bereitstellung von Rechenleistung spezialisiert ist.

Die aktuellen Entwicklungen im Bereich der Hardware-Virtualisierung zeigen, dass aktuelle Prozessoren leistungsstark genug sind, um mehrere verschiedene Betriebssysteme gleichzeitig ausfüren zu können. Die Benutzung von Hardware-Virtualisierung verursacht nur geringe zusätzliche Kosten.

Meine Arbeit zielt darauf ab, diese beiden Verfahren zu einer neuartigen Ausführungsumgebung zu kombinieren. Diese Umgebung wird die sichere Ausfürung von benutzerdefinierten Applikationen in virtuellen Maschinen auf einem entfernten System bereitstellen. Die Ausführung wird sich wie die Ausführung eines Batch-Jobs verhalten — schicke den Auftrag samt seiner Eingabedaten weg und bekomme die Ergebnisse zurück.

Contents

Preface i

List of Tables iv

List of Figures v

1. Introduction 1
- 1.1. Why use virtualization? . . . 2
- 1.2. The history of virtualization technologies . . . 3
- 1.3. Virtualization Techniques . . . 5
- 1.4. Problem Description . . . 10
- 1.5. Related products . . . 11
- 1.6. Goals of this work . . . 12

2. Requirements Analysis 14
- 2.1. Functional Requirements . . . 14
- 2.2. Non-functional Requirements . . . 18

3. Fundamentals 20
- 3.1. The Extensible Markup Language . . . 20
- 3.2. The Job Submission Description Language . . . 22
- 3.3. The Basic Execution Service . . . 25
- 3.4. Communication Model . . . 27
- 3.5. Secure Communication . . . 30

4. Design and Implementation 35
- 4.1. Overview . . . 35
- 4.2. The Xen-Based Execution Daemon . . . 36
- 4.3. The Xen-Based Execution Instance Daemon . . . 49
- 4.4. The Xen-Based Execution Command Line Client . . . 51
- 4.5. The Communication Protocol Stack . . . 51

5. Results 64
- 5.1. Execution examples . . . 65
- 5.2. Performance Analysis . . . 69

6. Conclusions and Future Work 74

A. Additional Background Information 77

A.1. Public-key cryptography 77
A.2. Calana 80

References **82**

List of Tables

3.1. XML Namespaces used in this work 21

4.1. Attributes of the `Error` message. 53
4.2. Important error codes and their descriptions. 53
4.3. Attributes of the `ConfirmReservation` message. 61
4.4. Attributes of the `StatusList` message. 61
4.5. Attributes of the `TerminateRequest` message. 62
4.6. Attributes of the `CacheEntries` message. 62

5.1. Test-environment machine configurations 65
5.2. Uncompressed *vs.* compressed small images 70
5.3. Uncompressed *vs.* compressed large images 71
5.4. Not cached *vs.* cached images 72

List of Figures

1.1. Virtualization architecture 4
1.2. Virtualization in the user-space 7
1.3. Para-virtualization architecture 8
1.4. Xen architecture 9
1.5. Preview of the XenBEE architecture 13

2.1. Basic execution semantics. 15
2.2. Batch job execution use cases. 15
2.3. Server deployment use cases. 17
2.4. UC Data Caching 17
2.5. Calana and XenBEE 18

3.1. A simple XML example 21
3.2. Basic BES Job-State-Model 26
3.3. Extended BES Job-State-Model 27
3.4. Protocol layers in an MQS-based communication. 29
3.5. Example MQS topology 29
3.6. Public Key Infrastructure 31
3.7. Secure communication with TLS 32
3.8. Secure communication with MLS 33

4.1. Overview of the XenBEE components 35
4.2. Components of the xbed 36
4.3. The job-model used in the XenBEE 39
4.4. Handling of *activity-objects* with an activity-queue. 40
4.5. Summary of executing a task 41
4.6. Start Instance Activity 42
4.7. File Retrieval Activity 43
4.8. Stage-In Activity 44
4.9. MSC List Cache Entries 46
4.10. Protocol Layers 52
4.11. MSC Message Layer Security 57
4.12. Message Layer Security 58
4.13. MSC Create Reservation 60
4.14. MSC Confirm Reservation 60
4.15. MSC Request Task Status 61

5.1. Experimental setup 64
5.2. Hello World example execution 66

5.3. POV-Ray execution result 67
5.4. Uncompressed *vs.* compressed small images 70
5.5. Uncompressed *vs.* compressed large images 71
5.6. Not cached *vs.* cached images 72

A.1. Architecture of Calana 80
A.2. Calana Job Model 81

"Virtualization is a paradigm shift; it changes how you think about your resources."

(Intel Corporation)

Chapter 1.

Introduction

Virtual machine (VM) technology is a major development in computer systems design [4]. By providing efficient "copies" of complete computer systems, it has extended the multi-access, multi-programming and multi-processing systems to be multi-environment systems as well [20].

Today's personal computer systems are powerful enough to provide virtualization technology which has long time been reserved for high performance mainframe systems only. The paper *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor* [49] discusses the problems of how virtual machines can be efficiently supported by the IA32 architecture.

In the last few years the remote execution of applications gained on importance, because powerful server systems need not be maintained locally anymore. They can be provided by different organizations in a centralized fashion. Grid middlewares such as Globus [19] or Unicore [59] are examples for execution environments in which users can send their computation to some remote computer.

In current grid environments it is always a problem of how an application is referenced by a user. To actually run an application on a remote resource, the application must be exactly identified either by some component of the middleware, or by the user itself.

The problem with that is that the application could be installed in different locations on each remote resource — if it is installed at all. If the user specifies the exact location he effectively limits the number of resources to which he can submit the job. If, on the other hand, the middleware defines the exact location and the user only specifies a logical identification, a mapping between those logical applications and the actual executables must be provided.

Another problem arises when a single computing resource is shared among several jobs. The middleware must make sure that jobs which are assigned to the same resource do not disturb each other. To make that clear, take two jobs from different users which are both scheduled to be executed on the same grid-resource. The least problem that can occur is that one task is unfair to the other one (*i.e.* it consumes too much CPU time or memory) — this can be handled by simply terminating the misbehaving process. A much worse problem is a misbehaving job that does so on purpose. Such a job could for instance attempt to steal confidential and private data from other jobs without being noticed by the execution system.

This work tackles these problems by providing the user with a secure execution environment that the user can setup himself. That means the user knows for sure that a particular application will be available, that the application is actually working and where it is located. Each provided

execution domain is running in its own dedicated virtual machine. That means, a task will not be able to access any data that belongs to a different task.

The next section will shortly survey the reasons why to use virtualization. After that an excursion to the past and current virtualization technologies is taken. And finally this chapter ends with a description of the goals of this work.

1.1. Why use virtualization?

The following sections discuss some benefits of using virtualization technologies instead of the conventional approach of using several stand-alone machines. The various virtualization technologies that are available nowadays are discussed in section 1.3. The following thoughts are based on the ideas and information that can be found in [3] and [54].

Application Development

To satisfy customer demands, not a single, general purpose operating system (OS) can be used. Over the time dozens of specialized systems have evolved — for instance on consumer desktop systems one can find Apple's MacOS, Microsoft's Windows or different Linux distributions, just to name a few. On cluster systems or dedicated systems which must provide outstanding security and availability other OSs are typically used.*

Each one of those operating systems has been designed to address the particular needs of a large segment of the marketplace [3], but it also imposes inconveniences for application developers for instance. An application that should be available on more than one OS has to be not only *adopted* to each abstraction layer an OS defines, but also *tested* on each.

The testing of such an application requires an actual installation of each target OS. The traditional approach to that was to simply take a dedicated machine. Virtualization however makes it possible to have each target OS installed on the *same machine* in parallel. The developer can have each one of these OSs right on his workstation.

Server Consolidation

Consider a small company that wants to take care of its web or intranet presence on its own. This requires at least three different server appliances: a DNS server, a web server and a database server that supports the latter. Three different solutions to this problem come into my mind.

- The first one is to take a single machine which runs all three appliances on top of one operating system. The problem in this case is that if just one of the appliances is compromised by an intruder, the whole machine is compromised as well.
- The next solution would be to deploy a single machine for each service, *i.e.* three physical machines in this case. This closes the security issues of the previous scenario, but it leads to increased costs for maintenance and administration.

* Linux may be an exception due to its high adaptiveness

- The third solution secures each service in its own virtual environment but on a single physical machine. This is the best of both worlds. It has the maintenance costs of the first solutions with a little overhead in administration, but the security of the second solution.

Virtualization in Grid-like Environments

A grid is a computing infrastructure in which many different resources are connected to each other by some kind of a network. The building blocks of a grid infrastructure are called "fabrics" [18]. The fabrics provide different kinds of resources to the user, such as computation resources, storage devices or instruments like a satellite dish.

Computational jobs that are submitted to a grid will eventually be executed on one of the computation resources. To maximize the utilization of computation resources, the same resource could be used to execute different jobs at the same time. Additionally, the acquired resource must be configured to support the application that is to be executed. To ensure the security of the involved computation resource, only trusted applications are allowed to be executed.

Virtualization technologies can in this case be used to allow the secure sharing of a given computation resource, while also supporting unknown applications.

The disadvantage of virtualization is that some overhead is imposed on the involved system. Actually, additional abstraction layers always come along with some overhead. Modern processors however are currently evolving to support virtualization techniques directly on the hardware.

The following section describes the history of virtualization technologies and why they have been developed in the first place.

1.2. The history of virtualization technologies

The history of virtualization starts with a paper entitled "Time sharing in large, fast computers" [56] written by Christopher Strachey in 1959. His idea bases on a single-CPU system which processes jobs one after each other. If a program blocks due to some peripheral access the next program in the queue gets started and will be run until the next peripheral access occurs and so forth. The system presents the user with a *logical CPU* and a scheduler assigns this logical CPU transparently for the user to a physical CPU. His concept of "time-sharing" is now known as *multi–programming* as Christopher Strachey states in a letter to Donald E. Knuth in 1974 [29].

This very simple scheduling strategy maximizes the utilization of the most worthy resource to that time — the CPU — and provides the base for current scheduling strategies such as multitasking.

The Atlas Project

Later on in the early 1960s, the "Atlas project" [24, 25] — a joint effort between the University of Manchester and Ferranti Ltd. has been founded. The Atlas computer has been the most powerful mainframe computer in the world in those days. It provided spooling mechanisms and pioneered in *demand paging* and *supervisor calls*, they also invented "virtual memory" — called "one-level store" in the Atlas system.

The supervisor calls where activated through interrupt routines or by so-called *extracode* instructions within an object program. Atlas made use of two "virtual machines" — one executing the supervisor and the other was used to run user programs.

The M44/44X Project

The IBM Watson Research Center has been the home for the M44/44X Project in the mid 1960s. The goal of this project was to evaluate the upcoming concepts of time–sharing [12].

The research team, led by R. A. Nelson, developed a way of partitioning an IBM 7044 machine into sub-machines that were each images of the 7044 with less memory — the main machine was called M44, the sub-machines 44X, thus the project's name.

Especially David Sayre and Belady made extensive experimental studies to evaluate the performance of virtual memory, load control and various scheduling policies [1, 12].

IBM virtual machines and the "virtual machine monitor"

IBM has perhaps been the most important force in the virtualization area. A number of IBM-based virtual machine systems were developed: the CP-40 (for a modified version of IBM 360/40), the CP-67 (for the IBM 360/67) and of course the famous VM/370, and many more [8, 36].

The VM/370 is the name for three operating systems, the *Control Program* (CP), *Conversational Monitor System* (CMS) and the *Remote Spooling and Communications Subsystem* (RSCS) [8]. Together they provide a way to form virtual machines, which can be used by many users. The CP therefore simulates multiple copies of the hardware on which it is running. The CMS is the operating system which runs in such a "virtual machine" and provides access for the users.

These virtual machines were typically identical "copies" of the underlying hardware [36]. A special component called the "virtual machine monitor" (VMM) ran directly on the real hardware (see Figure 1.1). Several virtual machines could then be created by using the VMM to assigning parts of the hardware to the virtual machine.

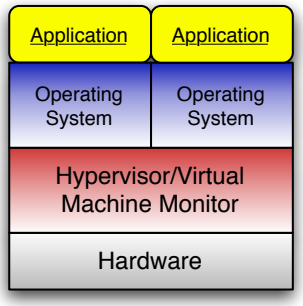


Figure 1.1.: The bare-metal virtualization: A virtual machine monitor runs as small "operating system" on the real hardware and provides access for virtual machines.

The virtual machine could then run an operating system on its own and had only access to those parts of the hardware that have been made available through the VMM. By this, new

operating systems could be developed and tested in a stable and secure manner. Actually, the VMM itself ran as a client to another VMM for debugging purposes.

This kind of virtualization is called "full" or "bare-metal" virtualization, because an operating system that runs in a virtual machine instance provided by the VMM actually thinks it runs on real hardware.

1.3. Virtualization Techniques

In contrast to the low-level virtualization that was used in the old mainframe computer systems, today's virtualization approaches are manifold. Virtualization is nowadays available in all abstraction layers that compose a computer system, *i.e.* not only on the hardware layer, but also on the operating system and application layer.

All virtualization technologies are backed up by the Church-Turing Thesis [5, 57]. The conclusion of this thesis is that every computer can simulate any other computer. A nice formulation of this thesis has been stated by Rowland [50]:

> *"Any real-world computation can be translated into an equivalent computation involving a Turing machine."*

The following sections provide a taxonomy of the currently available virtualization technologies. The sections increase the level of abstraction step by step, starting with the *partitioning* of the lowest layer, *i.e.* the existing hardware and ending with *para-virtualization*. The latter is based on executing machine code partly on the physical hardware and partly by emulating the behavior.

1.3.1. Partitioning

The first virtualization technique bases on the partitioning of the available physical hardware [3]. It is available since the 1960s. Two different approaches are available, a software based approach and a hardware based approach.

Software partitioning

The IBM CP/67 has been the first operating system which provided virtual machine support, it was running on the System/360 Model 67 and was first available in 1967 [3].

As described earlier, the CP gave each user a virtual machine on which the CMS (a single user operating system) was running and provided the user with command processing and information management functions. Each virtual machine was "copy" of the base hardware architecture, it was possible to run OS/360 in a virtual machine and "in fact, even CP/67 itself was run "second level" in a virtual machine for the purposes of debugging and testing" [3].

Hardware partitioning

Hardware partitioning is an enhancement over software partitioning and was introduced by the IBM *System/370 158 MP* and *168 MP* systems. In 1967, IBM introduced multiprocessor

versions of Model 65 and 67, which provided duplexed hardware to achieve tolerance against single hardware failures. By splitting up the whole system into two sides, two separate systems could be created which ran totally independent from each other.

1.3.2. Operating system-level virtualization

This kind of virtualization uses the same kernel for the host and all guest systems. To provide a guest environment with a virtualized system, additional support on the kernel level is required. The guest systems run within the host environment without knowing that they do so. Examples include Solaris Containers, FreeBSD and OpenVZ. All three provide *Virtual Private Servers* (VPSs), *i.e.* completely isolated server environments.

FreeBSD jails are used to shut a service or a special server application in its own environment. They prohibit an application from accessing data that is outside of the virtual environment which an administrator has reserved for this jail.

1.3.3. Application virtualization

The Java virtual machine [23] is a well-known example for this kind of virtualization. A program written in the Java language is compiled into a *Java byte code* and will be executed by the JVM upon execution. The virtual machine and the operating environment together are called the *Java runtime environment*.

To improve execution performance, an additional component called the *Java hot-spot compiler* translates small portions of often used code segments into machine code (a technique called "dynamic translation"), the translated code can also be cached and therefore be reused later.

1.3.4. Emulation

This kind of virtualization simulates a complete hardware architecture in all details. By that every operating system and therefore any application that has been developed for this particular architecture can be executed within the emulator. Some examples are:

- *Wine [62]* — Wine is not quite an emulator, but nonetheless I put it in this list, too. It emulates the Windows API, but executes many functions directly on the underlying x86 hardware without emulating each instruction.
- *Bochs [2]* — this is a very portable open source IA-32 PC emulator. Each machine instruction will be interpreted and is handled in software. This emulator may for instance be used to run x86 code on a PowerPC platform.
- *QEMU [42]* — this is both an emulator and an virtualizer, since it support two modes of operation. As an emulator each instruction gets interpreted as it is the case of *bochs*, *e.g.* it is possible to emulate an ARM processor on your PC. To achieve better performances a technique called *dynamic translation* is used. Running as a virtualizer, QEMU is able achieve nearly native performance, since most of the instructions are directly executed on the host CPU — to make this possible a kernel module (QEMU accelerator) is required.

The disadvantage of a pure emulation of a hardware architecture is the rather poor execution performance. Each emulated instruction has to be translated into at least two (but probably many more) actual machine instructions of the host architecture*.

1.3.5. Para-virtualization

Typically, the instructions (or most of them) of a virtual machine are directly executed on the underlying physical processor to get best performance results. Unfortunately, some instructions are simply "not designed" to be virtualized at all (see 1.3.6) — those instructions perform changes or query the state of some part of the physical hardware, but need not to be run in "privileged mode".

All instructions that cannot be virtualized have to be rewritten by the VMM before they get executed which means a loss in performance. VMWare [60] for instance runs as a stand-alone application on top of an operating system (supported by some extensions to the kernel) and uses this kind of binary rewriting. Figure 1.2 shows a schematic overview of such an architecture. The host operating system runs directly on the physical hardware, whereas the VMM is executed as an application that runs besides other applications on top of the operating system.

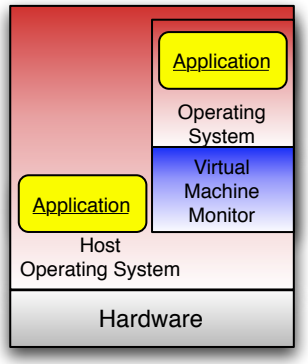


Figure 1.2.: Example for the VMWare virtualization software which runs as an application on top of some operating system

The Xen hypervisor [64] however, avoids the binary rewriting problem by providing its own "architecture". A guest operating system has to be ported explicitly to this architecture (see Figure 1.3) — an operating system which has been ported to the Xen architecture is also called "a Xen-aware OS". The proposed architecture contains only small modifications to the real hardware architecture, *i.e.* they mainly modify the non virtualizable instructions.

The term *para-virtualization* has first been used in the description of *Denali* [61]. In this case the VMM presents its virtual machines with a **nearly identical** copy of the underlying hardware, but the virtualized hardware is much less complex than the physical hardware (no BIOS, simpler devices, etc.). The modifications to the virtual hardware architecture require again that the operating system running in a virtual machine has to be aware of those modifications.

* Reading the instruction that is to be emulated and interpreting it cannot be implemented with a single machine instruction.

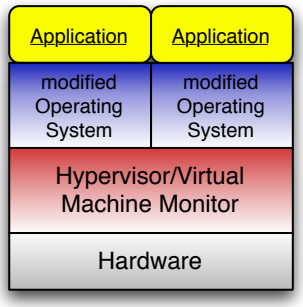


Figure 1.3.: An architecture that uses the para-virtualization technique

The disadvantage of the para-virtualization technology without binary rewriting is that legacy systems cannot be executed. These systems are often closed source which means that they might not be ported to the new architecture at all.

The next section describes the problems that arise when full virtualization (*i.e.* no binary rewriting or modifications to the architecture) is to be used with the x86 processor architecture.

1.3.6. Problems with Virtualizing the x86 Architecture

In [53] a hardware implementation of *protection rings* is described. These protection rings were required for the security of the MULTICS operating system. However, these *protection rings* are still present in today's processor architectures, especially in the x86 architecture.

Protection rings are used to separate "privileged" instructions from "unprivileged" ones. The x86 architecture provides a total of four rings, with ring 0 being the most privileged one — typically only two of them are used: ring 0 and ring 3. The former is occupied by the operating system kernel, whereas the latter is used to execute user programs. Popek and Goldberg [36] assume a processor architecture, that provides two modes of operation, supervisor and user — so the requirements which have to be fulfilled by an architecture to be virtualizable apply to the x86 architecture as well. They have identified three different kinds of instructions: privileged, sensitive and innocuous.

- *Privileged* instructions are those, that must be executed in supervisor mode and trap if executed in user mode. The x86 architecture has many of these instructions, but all of them raise a *general protection fault* if executed in user mode [49].
- *Sensitive* instructions are those, that "have a major bearing on the virtualizability of a particular machine" [36]. Generously speaking, they change the state of the hardware in some way without trapping.
- *Innocuous* instructions are all those which do not do any harm to the state of the processor from the VMM point of view.

Virtualization on the x86 architecture is typically implemented by running the VMM in privileged mode and the virtual machines in user mode. For this to be successful, all "sensitive" instructions [36, 37] must trap into the VMM, so that they can be correctly emulated.

It has been analyzed that all privileged instructions of the Pentium instruction set correctly trap

(*i.e.* they raise a "general protection fault") and can be handled by the VMM [49]. Unfortunately there are seventeen instructions which are **sensitive** and **unprivileged**, so they do **not trap**.

This is the main reason why full virtualization has not been implemented for the x86 architecture for a long time. Intel and AMD, the leading manufactures for x86 based processors, are currently developing hardware support for virtualization, so that unmodified guest operating systems can be run under a VMM.

1.3.7. The Xen hypervisor

> "The term "hypervisor" is applied to computer systems that present a very basic user program interface — one which is so nearly identical to a particular computer machine interface that an operating system intended to support such machines may serve as a hypervisor user program without software modification." [21]

Xen [64] is a virtualization technology to share a given physical machine using smaller virtual machines (VMs). Each of these VMs has their own main memory, file space, access to one or more virtual CPUs and everything else that is required to run an operating system (see Figure 1.4). Xen belongs to the hosted virtualization group, which means that the VMM still requires an operating system to run and does not represent a stand-alone operating system itself.

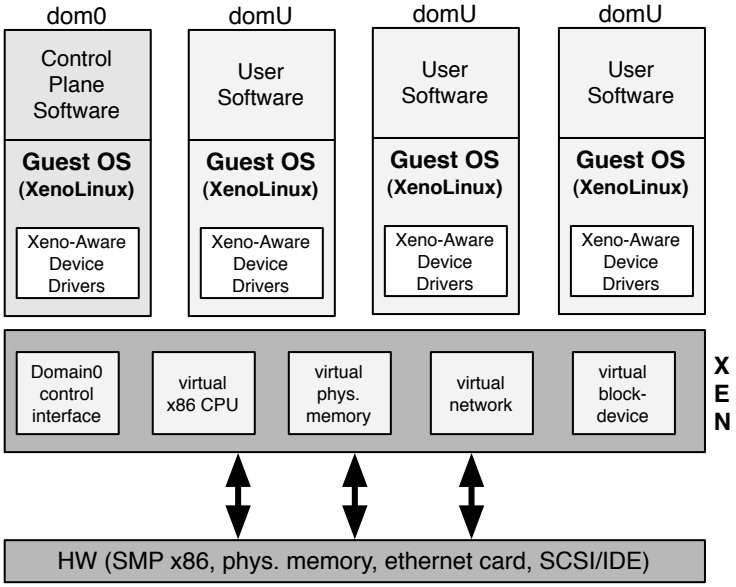


Figure 1.4.: The structure of a system running the Xen Virtual Machine Monitor and several user domains (taken from [15]).

Former versions of the Xen virtual machine monitor did only support para-virtualization. Each guest operating system had to be explicitly ported to the architecture provided by Xen. Since version 3.0 Xen supports the special hardware virtualization extensions developed by Intel and AMD, Intel-VT and AMD-V (or Pacifica) respectively.

Xen virtual machines (see Figure 1.4) are called "domains" and the top-level or most privileged

one is called `Domain-0` — or `dom0` for short — this is the one, which runs the control and management programs that are required to create new virtual machines. The virtual machine instances beside `dom0` are called "user domains" — or `domU`s for short — they are less privileged and their access to the hardware is controlled and managed by the hypervisor running in `dom0`.

The operating system which is running within a user domain accesses virtual hardware that is provided by the Xen-architecture, *e.g.* SCSI or IDE controllers to access virtual hard drives, network interface cards, virtual CPUs, graphic card and so on.

The following, final sections of this chapter define the problem space of this work. At first the problem of executing user-defined jobs with virtual machines is illustrated by the use of example scenarios. I then will point you to already available products which are similar to the proposed execution environment. Finally I will summarize the goals of this work.

1.4. Problem Description

The following two case scenarios are meant to introduce you into the *Xen-Based Execution Environment*. The first scenario represents a typical execution of a batch job. The second scenario describes the problem of on-demand server deployment. This is a technology that will be provided by grid environments in the future, but is still uncommon to current grid environments. It describes the process of deploying a service to key-locations within a network.

1.4.1. Batch job execution

Consider a user who wants to execute the POV-Ray raytracing program [39] on some remote computation resource. The input to this computation is the definition of the scene that should be rendered. The output of the program is a picture that shows the rendered scene.

To submit this job to a grid environment, the user has to define the executable (*i.e.* `povray`) that he wants to use. This could be done in a variety of ways:

- Each computation resource that is connected to this particular grid has the application installed at the very same location.
- The user specifies the application indirectly, *i.e.* he refers to a specific application by a logical name. The grid middleware that is responsible for the actual execution (*e.g.* a daemon that runs on the computation resource) must map these logical names to the actual executables.
- The grid provides an information service that can be queried whether an application is installed on a particular computation resource.
- The user passes the executable along with her job description to the execution host. This could result in security problems, because the application is possibly malicious.
- Only platform independent applications (*e.g.* Java applications) are supported.

This list shows that many different solutions are possible just to specify the application that should be executed. The *Xen-Based Execution Environment* (XenBEE) approaches this problem by putting the application itself into an *execution container*.

An execution container consists of a stripped-down operating system installation and the application that is to be executed. The container can be made available either by the user, or by some provider. This container will then be used by the XenBEE to create a Xen virtual machine instance that is dedicated to the execution of the user's application. Prior the execution can start the input data for the application has to be made available. After the execution has finished the output data has to be made available to user.

1.4.2. On-demand server deployment

Consider for example a group of people that wants to play an on-line multi-player game. Such on-line games tend to require a rather low latency network connection so that the game can be played without having lags. If I ran that server on my DSL connection with many connected users, there would be large transmission delays. This is because the DSL connection simply does not provide the required quality of service, *i.e.* low latency.

The provision of an on-demand server deployment process could be used to run the game server on a well-connected remote host. The game server must be available within a relatively short amount of time and needs to stay available for just a couple of hours. The hosting environment for these kind of servers could be provided by a rental company that has specialized in such services.

The XenBEE could be used to deploy such a server. The server would be contained in a previously prepared *execution container*. The game server will then run in a virtual machine that is configured with an appropriate amount of main memory, computing performance and network accessibility.

1.5. Related products

The following two available products provide both the possibility to deploy a virtual machine in a remote location. To my best knowledge, they "just" provide the deployment of virtual machines. They do not provide batch job execution semantics on their own.

1.5.1. Amazon EC2

The *Amazon Elastic Computing Cloud* (EC2, [16]) is part of the *Amazon Web Services* project and provides a web service for remote execution of user provided virtual machine images.

Amazon provides you with a personal storage area where you can deposit arbitrary data (such as *Amazon Machine Images*). Another Amazon web service is used for this: the *Amazon Simple Storage Service* (S3).

With the files that are located in your personal storage area, you can create as many virtual machines as you like. Through web service calls or by using one of the various command line tools, you can start, terminate and monitor your deployed instances.

The provided environment is very sophisticated, since it provides a completely secure storage area for possibly confidential data. It also provides fast and on-demand deployment of the virtual machine instances.

1.5.2. The XenoServer Open Platform

The XenoServer Open Platform is an implementation of the *global public computing* paradigm [26]. It provides the execution of *any code* from *any user anywhere*.

Servers *advertise* themselves and clients *select* the servers on which their computations or services shall be deployed. After selection of the servers, a client send the deployment specification to each one of the servers. A server may then accept or decline the request according to local policies or resource requirements.

To integrate this environment into a grid, one has to install the grid middleware in the deployed virtual machines.

1.6. Goals of this work

The previous sections have outlined the problem domain of this work. There is currently no product available which provides batch job execution semantics based on virtual machine deployment. One of the goals of this work is to provide such a semantic. The following list summarizes the goals of the *Xen-Based Execution Environment*.

- **Batch job execution semantics.** The execution environment must provide support for the execution of batch jobs. That means a user must have the possibility to submit new jobs, as well as monitor and abort his running jobs. The execution of batch jobs requires that arbitrary input data must be made available to the job, as well as generated output data must be made available to the user.
- **On-demand server deployment.** The execution environment must be able to create virtual machine instances that are reachable through a network connection.
- **Integrability.** The execution environment should be integrable into existing (grid) environments. That means standard or future technologies which are used in grid environments must be supported. This regards the description, status query and termination of jobs.
- **Security.** Since the execution environment could be used by many different users at the same time, it must provide a secure execution context to user and provider. The secure context includes the interaction of a user with the execution environment, as well as the execution of a job. From the provider's point of view the security context includes authentication and authorization of the users, as well as a secure communication.
- **Efficiency.** The usage of virtual machines should not impose much overhead on the total execution time of submitted jobs.

Architecture preview

The picture in Figure 1.5 shows a preview of the architecture that has been designed and implemented in this diploma thesis. In the first step the user makes his virtual machine image available on some server. The second step depicts the submission of the execution request. Therefore a client and a server application are involved, *xbe* and *xbed* respectively. Both are communicating with each other by the use of a message-queue server. The execution container

is retrieved by the *xbed* in the third step — from either a storage server, or a data cache. The job is executed in its own virtual machine in the fourth step, whereas the execution is under control of the *xbeinstd* component.

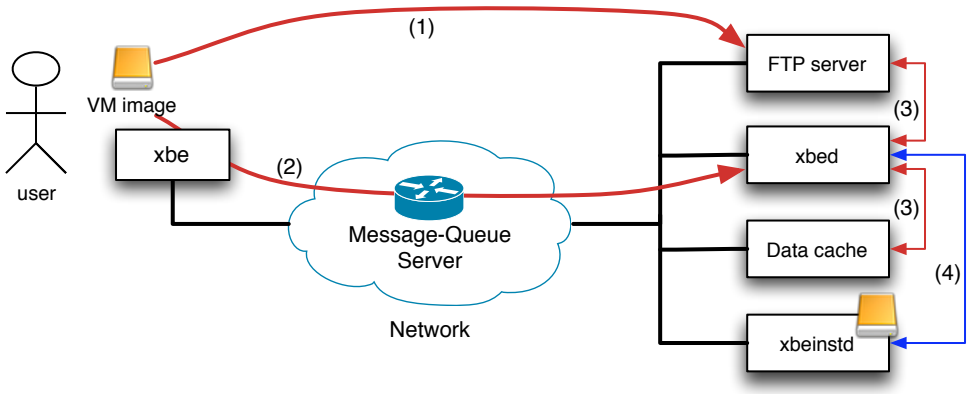


Figure 1.5.: Preview of the XenBEE architecture

Structure of the following chapters

The remaining part of this work is outlined as follows. In the next chapter, the requirements to the XenBEE are analyzed. Subsequent to that some basic technologies which have been used to implement the execution environment are presented. The fourth chapter describes the design and implementation of the proposed work. The results of performed experiments are provided in the fifth chapter. The conclusions of this work and some directions for future developments can be found in the very last chapter.

"Computers can figure out all kinds of problems, except the things in the world that just don't add up."

(James Magary)

Chapter 2.

Requirements Analysis

This chapter details on the goals that were outlined at the end of the previous chapter. To analyze the requirements to the XenBEE the analysis process is divided into two parts: *Functional Requirements* and *Non-Functional Requirements*.

The section on *Functional Requirements* aims to analyze the first two goals of this work, *i.e. Batch job execution semantics* and *On-demand server deployment*. Both are pure functional requirements that have to be designed and implemented in the XenBEE. This section will also contain some additional use cases the are related to the job execution. The provided use cases are analyzed with regard to *Integrability* of the XenBEE into grid-like environments.

The *Non-Functional Requirements* address the goals *Security* and *Efficiency*. Therefore some ideas will be presented that provide a secure and efficient execution of jobs.

2.1. Functional Requirements

This section discusses the execution semantics that are to be supported by the XenBEE. In particular it analyzes how batch jobs can be executed on a remote virtual machine and how server applications can be deployed on-demand to virtual machines.

The following sections describe the execution of these kind of jobs, but first of all the basic execution semantics are analyzed.

2.1.1. Basic execution semantic

Suppose you wanted to execute a job on a remote resource. The execution environment should at least provide the following functions: submission of a job, status retrieval and termination of a submitted job (see Figure 2.1).

In the XenBEE a job is defined by an *execution container* along with a description of the job. The execution container is a virtual machine image that contains the application to which the job refers.

The *query status* use case requires the modeling of a finite state automaton that describes the current state of a job (job-state model). The OGSA-*Basic Execution Service* [33] provides a stable, generic and extensible specification for such a job-state model (a detailed description can be found in Section 3.3 on page 25). To support the integrability with grid-like environments this specification should be used in the XenBEE.

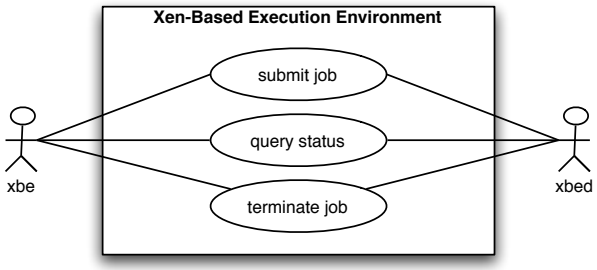


Figure 2.1.: Basic execution semantics.

The *terminate job* use case must always be available to a user. That means a user must be able to terminate the execution of a previously submitted job at any time. The *terminate job* use case is also included in the BES state model.

2.1.2. Batch job execution

A *batch job* is a program that is executed by a computation resource without further user input, *i.e.* the opposite to *interactive job*. Batch jobs typically transform input data into output data, whereas the input data may also be absent. If no programming errors have been made these kind of jobs finish after an undefined but finite time. The execution of the POV-Ray raytracer [39] to render a user-supplied scene is an example for this kind of jobs.

Figure 2.2 shows the individual use cases that are involved when submitting a batch job to the XenBEE. The user has to provide the virtual machine image and the input data. The *xbed* must then access these files to create a virtual machine that executes the batch job. The generated output data has to be made available by the *xbed* so that the user can access it.

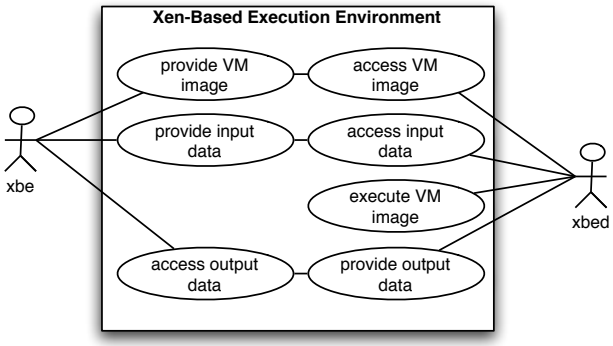


Figure 2.2.: Batch job execution use cases.

The following sections specify the requirements for the job submission description and provide some ideas on how an implementation could implement the data access.

Job description

The crucial part of this use case is the description of the job submission. In the past each grid middlewares such as Condor [6], Unicore [59] or the Globus Toolkit [19] used their own propri-

etary submission language. This made interoperability between different grids middlewares very difficult.

The *Job Submission Description Language* (JSDL, [22]) is generic description language for the submission of computational jobs to a remote resource. To the time of the writing of this work the mentioned grid middlewares have already moved to this language or are in progress to do so.

Since the XenBEE should be integrable into grid environments, a fixed requirement for the XenBEE is to use the JSDL. For more information on the JSDL consult Section 3.2 on page 22.

Selecting a VM image

The submission of a task includes the selection of an image that contains the application the user wants to execute. A sophisticated process of image-selection can be rather complicated, since it involves matching of available images against a description provided by the user. Such selection mechanisms are out of the scope of this thesis.

File provision and access

In a preliminary step the client must make the VM image, as well as the input data available to the *xbed*. The JSDL supports *Uniform Resource Identifiers* (URI, [44]) that can be used to accomplish this task, *i.e.* the user specifies the location of an input file with a URI. The *xbed* is then able to access (retrieve) the files. The same can be applied for the provision of generated output data, *i.e.* the user specifies the target location to which an output file should be uploaded.

Virtual machine creation

For each submitted job a new virtual machine has to be instantiated. This virtual machine uses the VM image provided by the user. The application that is to be executed is specified by the client with the use of the JSDL.

To actually execute the application in the virtual machine another component is required: the *xbeinstd*. This component is then used to control and monitor the execution on the virtual machine.

2.1.3. On-demand Server deployment

A *server application* or a *service* is a remotely executed program loops over the following steps indefinitely often: wait for user input, execute a computation on the input data, generate output data. A web server is an example for such an application: it waits until a user (or some service) makes a request to it, handles the request (*i.e.* retrieval of a document) and eventually returns the result (*i.e.* the document).

The use cases that are involved in the *on-demand server deployment* process are depicted in Figure 2.3. This process shares most of the involved use cases with the batch execution process.

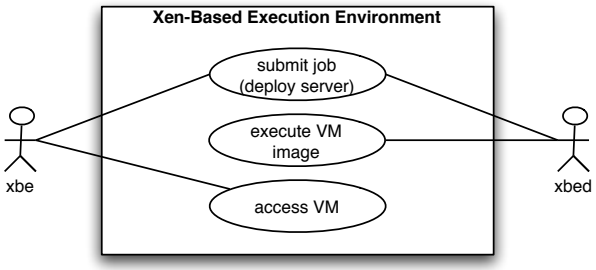


Figure 2.3.: Server deployment use cases.

In contrast to a batch job submission, the virtual machine instance may run "forever", *i.e.* the "job" runs until the VM is shut down by the user or the job itself is terminated. This must be explicitly stated in the job description.

Another difference is that the VM must be reachable through a standard network connection (*e.g.* TCP/IP connectivity). This can be the case for virtual machines that execute batch jobs, too, but it need not to.

Based on the network connectivity, a login to the VM can also be provided. For example by using the *Secure Shell* (SSH, [34]).

2.1.4. Caching of data

Imagine a user who wants to execute the same application several times. That would mean he has to submit the same image over and over again. This imposes a heavy load on the network that is connecting the user and the server. It would be wise to provide a caching mechanism, that allows the user to store his image on server-side. According to the *Locality Principle* [13], the caching should decreases overall execution time, too. The involved steps to cache data are shown in Figure 2.4.

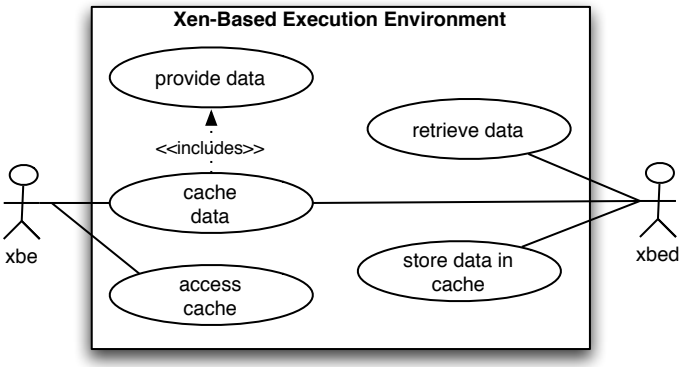


Figure 2.4.: A user who is requesting the caching of (arbitrary) data.

The *cache data* use case can also make use of URIs to refer to the data. The *access cache* use case requires that the entries can be *listed* and *referenced*. The entries should be specified as URIs, too. This makes them available for job submissions.

To provide shorter execution times, the XenBEE should provide a cache with the following requirements: Arbitrary data must be addable, a cache listing must be available, cache entries must be indentifiable by URI.

2.1.5. Support for Calana

Calana is a new agent-based Grid scheduler that uses auctions to schedule job submissions. A short description of Calana can be found in Appendix A.2 and in [10, 35].

Calana assumes that a computation resource supports reservations. That means the resource must provide semantics to *make*, *confirm*, *use* and *cancel* reservations.

Figure 2.5 describes how a user would interact with a system, that uses Calana for job scheduling and the XenBEE as a computation resource. This scenario requires an agent that implements the Calana protocol on the one hand and the protocol used in the XenBEE on the other hand.

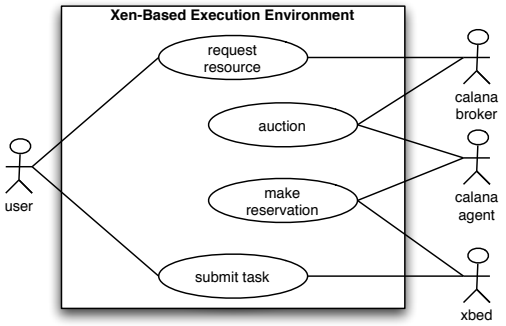


Figure 2.5.: The actors and use cases that are involved when a Calana agent uses the XenBEE as its resource.

The user requests a resource from the broker which will in turn open up an auction among its agents. One of those agents are shown in the figure. In order to bid on the auction the agent creates a reservation on the *xbed*. If the auction is lost, the reservation is automatically canceled. If the auction was won, the user is eventually presented a unique identifier for his reservation.

This reservation can then be used to submit a job to the *xbed*. The *xbed* must then check the validity of this identification number.

To support Calana the *xbed* has to provide reservation semantics, *i.e.* it must be possible to *make*, *confirm*, *use* and *cancel* reservations.

2.2. Non-functional Requirements

The following sections describe shortly which non-functional requirements the XenBEE should support.

2.2.1. Security

This section aims on requirements that are related to security. In particular three requirements are presented: *authentication and authorization*, *secure communication* and *secure execution*.

Authentication and authorization

Since the XenBEE provides a *service*, the provider of this service may want to restrict access to a selected group of *authorized* people.

Before granting a user the access to the execution environment, the identity of that user has to be verified, *i.e.* the user must be *authenticated*. Without authentication an unauthorized person could simply pretend to be an authorized person.

Secure communication

Secure communication between the *xbe* and the *xbed* is required to prevent *eavesdropping*, *tampering* and *message forgery*.

That means an attacker must not have the possibility to overhear probably confidential data that maybe included in a user's job description, nor should it be possible that he can modify or even create new messages that seem to come from this user.

A typical approach in Grid middlewares such as Globus [19] or Unicore [59] is to use public-key certificates (*e.g.* X.509 certificates) to provide authentication, authorization and secure communication. The XenBEE should follow the same principles. Section 3.5 on page 3.5 describes in detail how these requirements can be provided in the XenBEE.

Secure execution

This requirement targets at the actual job execution. The use of virtual machines provide already that the jobs cannot harm each other, because they are completely separated from each other.

But security has to be provided on the Xen-host as well. That means that data that belongs to one job (VM image, input data, output data, and so on) must not be modifiable or accessible by any other jobs — even if both jobs belong to the same user.

Since all files are accessed by publicly reachable URIs, to ensure the security of these files it could be possible to provide them encrypted. Before they can be used by the *xbed* to create a virtual machine, they have to be decrypted somehow.

2.2.2. Efficiency

Efficiency in the context of the XenBEE means that the overhead which is imposed by the communication and the use of virtual machines should be kept minimal.

The caching of virtual machine images can be used to decrease the time that is needed to deploy a new virtual machine. This affects both the batch job execution and the on-demand deployment of servers to key locations.

"A journey of a thousand miles begins with a single step."

(Lao-tzu)

Chapter 3.

Fundamentals

This chapter provides you with the description of basic principles and concepts that have been used in this work. The chapter is basically separated into two parts: Concepts that have already been pointed out in the previous chapter and descriptions of technologies that lead to important design decisions.

The first part starts with an introduction into the *Extensible Markup Language*. XML is the description language that is used for the *Job Submission Description Language*, the *Basic Execution Service* and the messages which are used in the implemented communication protocol (see Section 4.5 on page 51).

The second and last part of this chapter provides the reasoning which lead to the decision to base the XenBEE on an *asynchronous message-passing communication model*. It also discusses how the communication between the distributed components can be secured.

3.1. The Extensible Markup Language (XML)

The Extensible Markup Language [65] is a simple, yet very flexible, plain text based description format. The format represents a subset of the *Standard Generalized Markup Language* (SGML). It can be used in variety of ways and even more usages are discovered still. Usages of XML can be found in *XHTML*, *RSS*, *Atom*, *Math-ML* and many more. Due to the structured semantics of XML, more and more file formats are nowadays based on XML, thus replacing the old INI or Unix `rc` files — a very popular example in this field is the *OASIS Open Document Format for Office Applications*[*].

An XML-file is an ordinary plain text file, that could have been created by any text editor. The most important building blocks of XML-files are *elements*, *attributes* and *text*.

Elements are *logical structures*, that can have additional attributes and sub-elements or *children*, whereas the children can either be other elements or text. The following example shows you a small XML document. Every XML document contains **exactly one** designated *root* element, which is simply the first element in the document.

For parsing purposes, an XML document can be represented as a tree, this is shown in Figure 3.1. A widely used model for the in-memory representation of XML documents is the *Document Object Model* (DOM). Most of the available XML parsers, provide an interface for parsing

[*] More information about the Open Document Format can be found on `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office`

a document into an in-memory representation that follows this model. The programmer can then simply add, delete or modify elements and attributes by using an object-oriented interface. Using the example in Figure 3.1, a programmer could for instance iterate over all children of the `root`-element, that have a tagname (*i.e.* the name of the element) equal to "child" — in this case, that would result in just two elements.

```
<root foo="bar">
  <child/>
  Some text
  <child>More text</child>
</root>
```

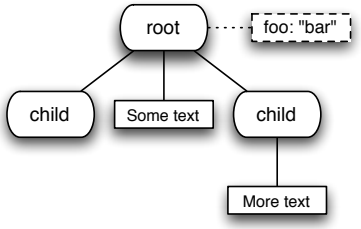


Figure 3.1.: A simple XML example

3.1.1. Namespaces

XML documents may contain elements and attributes from different vocabularies (*i.e.* different document types). To resolve ambiguity between the involved vocabularies, the W3C recommends the use of unique *Namespaces* that are assigned to each element. A document may contain a *default*-namespace to which all elements belong that do not have a special namespace assigned. Within a single document, namespaces are given a logical name. The logical name itself is assigned the unique Namespace identifier (*e.g.* a URI). Some namespaces and common "names" for them are given in the following table (Table 3.1):

logical name	namespace URI
`xsd`	`http://www.w3.org/2001/XMLSchema`
`jsdl`	`http://schemas.ggf.org/jsdl/2005/11/jsdl`
`jsdl-posix`	`http://schemas.ggf.org/jsdl/2005/11/jsdl-posix`
`dsig`	`http://www.w3.org/2000/09/xmldsig#`
`bes`	`http://schemas.ggf.org/bes/2006/08/bes-activity`
`xbe`	`http://xenbee.berlios.de/schemas/xbe/2007/01/xbe`
`xbe-sec`	`http://xenbee.berlios.de/schemas/xbe-sec/2007/01/xbe-sec`
`xsdl`	`http://xenbee.berlios.de/schemas/xsdl/2007/01/xsdl`

Table 3.1.: Important namespaces

Namespaces are specified in the XML using the special attribute `xmlns`. An attribute of an element, which reads `xmlns="www.example.com"`, sets the default namespace to the given URI, while `xmlns:foo="www.example.com"` makes the same namespace known as the logical name "foo". In another document the same namespace could have been assigned the logical name "bar" as well.

To specify that a given element belongs to a namespace other than the default namespace, the element's name is prefixed by the logical name of the namespace, *e.g.* `foo:child` means, that the "child" element belongs to the namespace defined by "foo".

3.1.2. Validation of XML documents

A really nice and very useful addition to XML is the possibility to *validate* an XML document. There are two mechanisms to provide validation, the *Document Type Definition* (DTD) and *XML-Schema*.

Document Type Definition

A DTD defines for a particular document what elements are allowed and how their attributes look like. The composition of elements to form container (*i.e.* parent) elements can be described in a rudimentary way. Unfortunately, the DTD uses its own syntax, that has nothing in common with the syntax of an XML document. For an author of an XML document type* that means in particular, that he has to learn two different syntaxes.

XML-Schema Definition

XML-Schema is itself defined using XML as its description language and obsoletes the DTD. It is much more powerful, for instance, an author is able to restrict the actual data an element or attribute may contain. Let's for instance say, a given attribute can only take on non-negative integers. To reflect this constraint in the schema definition, the author would set the type of the attribute to `xsd:nonNegativeInteger`.

There are many predefined data types, an author may use to create new data type constraints. An XML-Schema validator complains, if a document, that is supposed to conform to that schema, contains the just introduced attribute with a negative value, *e.g.* -1.

The advantages of XML-Schema over a DTD are obvious. When using DTDs, the application itself was responsible to check the validity of each XML document that it used. That means, the same functionality had to be implemented over and over again, *i.e.* each time a new application wants to make use of a given XML document type. While using XML-Schema definitions, the author of an XML document type defines the validity constraints just once and any application may rely on that.

To make that clear, here is a short example: Suppose there is a definition for an element called "`entry-id`" which may only take on positive integer values. Since a DTD does not support constraints on data types, each application must check for itself if the value matches that type. Now suppose the constraint for that element is changed so that the number 0 is included as well — in each application, the validation code must be modified to match the new constraint.

3.2. The Job Submission Description Language (JSDL)

JSDL is a very extensible XML-based description language for the submission of computational jobs [22]. With JSDL you are able to describe all requirements that a computational job may

* for example the configuration file format of an application

```
<jsdl:JobDefinition>
 <jsdl:JobDescription>
    <jsdl:Application>
      <jsdl-posix:POSIXApplication>
        <jsdl-posix:Executable>
          /bin/echo
        </jsdl-posix:Executable>
        <jsdl-posix:Argument>Hello World</jsdl-posix:Argument>
      </jsdl-posix:POSIXApplication>
    </jsdl:Application>
  </jsdl:JobDescription>
</jsdl:JobDefinition>
```

Listing 3.1: A small "Hello World"-example written in JSDL

need for the submission to a remote resource — mainly the JSDL addresses grid resources but it is not limited to that.

Nearly every element of the JSDL specification may contain arbitrary many user-defined elements from other XML specifications. Therefore is the JSDL adoptable to upcoming user requirements. The JSDL has been extended in this work to support the description of virtual machines (see Section 4.2.4 on page 47).

To explain this, consider the following example. To execute a job on a resource a previously acquired reservation is required. To add this information to the job submission, the JSDL would include an additional element which holds all information regarding the reservation. Such extension elements are purely optional and systems that are unaware of a particular extension element may just neglect it.

The following section roughly describes the most important components that are needed to form a useful submission description.

3.2.1. "Hello World" with the JSDL

A JSDL document does always start with the `JobDefinition` element, which is the top-level element and holds all required information about the job.

Let's assume a user wants to execute a small program on a remote resource. The program will indeed be very simple, it just prints the string "`Hello World`" to its standard output stream. The execution of this application on the user's local computer could be similar to:

```
$ echo "Hello World"
Hello World
$
```

This excerpt represents the execution in a standard UNIX command shell. Note that the `echo` program does nothing more than writing the parameters passed to it to its `stdout` stream. Now suppose the user desires to execute the same program on a remote resource. The JSDL

document will look somewhat like the document shown in Listing 3.1.

Note, that the shown JSDL document describes exactly the same execution the user had previously performed locally. The executed `echo` program again writes its arguments to its `stdout` stream. Different from the local execution is in this case, that the output will be lost, since the user did not specify what should be done with the generated output. If the user was interested in the program's output, he had to specify the redirection of the output stream to some file and a staging operation that transfers the created file to some location he has access to.

3.2.2. Important elements

A typical JSDL document consists of the following parts — job identification, application description, resource descriptions and data staging elements. To keep the example simple, only the second one has been used in the example in Listing 3.1.

Job identification

The `JobIdentification` element is used to hold information about the job, such as a descriptive name, that is mostly interesting for human beings. Nonetheless it may hold additional information that could be of interest to applications processing the document — such as annotations (*e.g.* a unique task identification number could be stored in such an annotation).

Application

With the `Application` element, a user describes the program itself — *i.e.* the real executable, that is going to be used. A special extension — `POSIXApplication`, also defined in the specification [22] — can be used to describe executables for POSIX-compliant operating systems [38]. You have already seen the usage of this extension in Listing 3.1, where it had been used to specify the execution of the `echo` command line program.

Resources

This element can be used to describe various resource requirements of the application. Some of the many resource types one can use are listed below.

- the number of CPUs the job requires
- the operating system required by the job
- amount of virtual memory that must be available for the job
- file-systems and their expected mount-points

All specified file-systems must be made available for the job prior execution. Every file-system specification defines a unique name that can be used to refer to that particular file-system in other elements, such as staging operations. Thereby the user can define *logical names* for special directories within the execution environment of the task.

Staging operations

The `DataStaging` element is used to define staging operations. These operations can either be *Stage-In* operations, which refer to files that have to be transfered into the execution environment prior the execution of the task, or *Stage-Out* operations, which refer to transfers that have to be made after the task has been executed.

A user may specify the `DataStaging` element as often as he likes to. The most relevant elements within a staging instruction are the `Source` and `Target` elements, both of them can hold a `URI` element to specify a generic location. The mandatory `FileName` element points to an actual file within the file-system hierarchy of the execution environment of the task. The actual location of a file can be given relative to a previously defined file-system, in this case the `FilesystemName` element must be specified and is required to contain the logical name of `FileSystem` resource.

Conclusions

The JSDL is a powerful description language for computational jobs. It aims to cover the description of the job submission for typical computational jobs. The XenBEE is such an example, since the JSDL does not know anything about executing jobs on virtual machines. The extensions to the JSDL that I have designed and implemented are covered in Section 4.2.4 on page 47.

The next section deals with the *Basic Execution Service*, a specification that provides a common execution semantic of computational jobs. This semantic is backed up by an extensible state-model to describe the job execution.

3.3. The Basic Execution Service (BES)

The *Basic Execution Service* [33] is a specification that defines a service (*e.g.* a web service) which provides functionality to control *Activities*. An *activity* can be seen as an abstract view on a computational job. A user is able to submit an activity to the execution service and can later on control and monitor that activity — using web service calls, for instance.

Control of the activity is basically limited to a request which *terminates* the activity. The monitoring of an activity results in returning the activity's current state.

The state of an activity is modeled using a finite state automaton. The specification of the BES incorporates a simple, but very extensible, state machine for activities. It comprises a total of just five states an activity can be in at any time: `Pending`, `Running`, `Finished`, `Failed` and `Terminated`.

To be extensible and integrable into existing environments, the states of the BES are represented using a XML specification. The element which represents the current state of an activity is named `ActivityStatus` and belongs to the *bes* namespace as it has been defined in Table 3.1, the state itself is then specified using the `state` attribute:

```
<bes:ActivityStatus state="Running"/>
```

The basic state machine, or state-model, is shown in Figure 3.2. To reflect the request for termination of an activity, each non-terminal state provides an outgoing transition to the `Terminated` state.

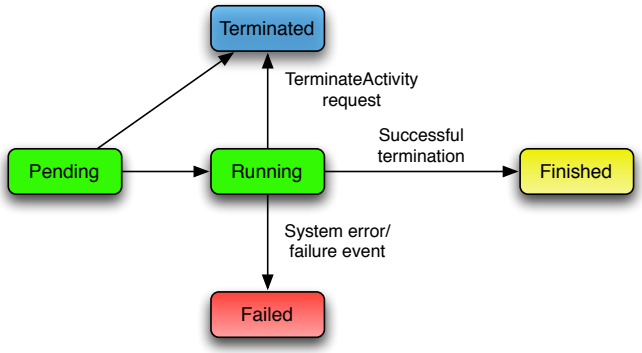


Figure 3.2.: This is the job-state-model as it is defined in the BES specification [33]

This basic state-model represents everything a possible client of the execution service needs to know. An actual execution service may require to use additional states. It can define both new states and new transitions, as long as it conforms to a rather simple rule: the **visual behavior**, as it is experienced by some client, must not be altered. A breakage of this rule would be the introduction of a transition from the `Running` state back into the `Pending` state. Clearly, the visual behavior a client experiences, has changed, since the client simply does not expect that the activity is suddenly in `Pending` again.

3.3.1. Extending the state-model

New states can be added by splitting up either one of the "basic" states, or a state that is an extension itself. Among these sub-states any number of new transitions may be introduced. The following example for an extension has been taken from the BES specification [33].

Suppose the execution service provides a way to suspend a given activity. This requires not only additional user requests — one to request *suspension* and one to request *resumption* — but also two new states. These states are modeled as sub-states of the `Running` state, since an activity may only be suspended while it already running. The `Running` state is now internally split into the sub-states `Executing` and `Suspended`. The extended state machine is shown in Figure 3.3

The nice thing about the extensibility of this state-model is that any client that "understands" the basic model, will understand any extended model as well. That is because the *visual behavior* does not change and therefore will never "anything unexpected" happen. This visual behavior is directly reflected in the XML specification of the current state. Any additional states are represented by user-definable elements added to the `ActivityStatus` element as sub-elements. Let's assume the just modeled extension defines its own state representation using its own namespace. The current state of a suspended activity could then be written as:

The state is still `Running`, but any client that is aware of this extension will know how to

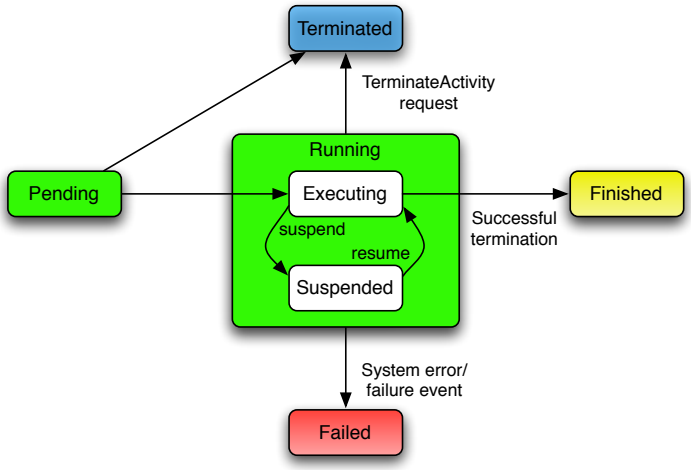


Figure 3.3.: The basic state-model has been extended to support suspension (taken from the BES specification [33]).

```
<bes:ActivityStatus state="Running">
   <ext:Suspended/>
</bes:ActivityStatus>
```

interpret the `ext:Suspended` sub-element, *i.e.* it could provide a user with the possibility to resume the action.

The state-model that is used by the XenBEE extends the basic model with support for staging operations, reservations and virtual machines, it can be found on page 39 in Section 4.2.1.

This section closes the description of the technologies that were required for the XenBEE to provide batch job execution semantics and integrability into grid-environments. The following sections motivate the applied communication model and how this model can be enhanced to provide secure communication.

3.4. Communication Model

Communication in distributed systems can be either *synchronous*, or *asynchronous* [30]. This section summarizes these two models and results in the definition of the communication model used by the XenBEE.

3.4.1. Programming Models

The programming model of synchronous communication is called *Remote Procedure Calls* (RPC, [51]). It provides the same function call semantics as a local function call, *i.e.* the program waits until the result is computed and returned by the remote system. Commonly used implementations of this model are the *Common Object Request Broker Architecture* (CORBA, [7]), the *Remote Method Invocation* (RMI, [48]) or the *Distributed Component Object Model* (DCOM, [11]).

Asynchronous communication follows the emerging paradigm of the *event-based communica-*

tion model [30]. A request that is sent to a remote system does not have an immediate result. The result, if any, is received by the caller asynchronously to his current computation. The distributed components that use asynchronous communication are interconnected by using message-passing technologies such as the *Message Passing Interface* (MPI, [32]).

The XenBEE can be implemented using either of two models. Consider for example the request for the termination of a submitted job using a single-threaded client application. The client could make a remote procedure call and wait until the termination has been performed on the remote site. Or the client just sends the request to the server and is able to accept further input from the user.

The XenBEE will be designed to use the asynchronous communication model. This implies the use of a message-passing technology.

Erzberger and Altherr [17] state, that:

> *"Every DAD (Distributed Application Developer) needs a MOM (Message Oriented Middleware)"*.

3.4.2. Message Oriented Middleware (MOM)

Message-passing in distributed systems need not be based on direct (*e.g.* TCP) connections between each component. The messages can easily be transmitted to an intermediate system which forwards the message to the target system. This section describes a middleware component called *Message-Queue Server* (MQS) which can be used in this kind of distributed systems to improve communication quality.

The abstraction from direct connections between each component leads to the definition of *logical connections*. A logical connection is a connection between distributed components that pass messages to each other while not being directly connected to each other.

To send a message to some component using logical connections, the message is addressed to a logical destination and sent to an intermediate server that hopefully knows the actual target. To receive messages from other components, a component registers itself with the intermediate server.

Such an intermediate server is a *Message-Queue Server* (MQS). The protocol layers that are involved when application data is to be transmitted from one component to another are depicted in Figure 3.4.

The usage of such an intermediate MQS has several important advantages over direct connections between the distributed components:

- All messages are sent to **logical queues** (*i.e.* end-points). This means that the details of the connection of a remote component is hidden to other components. For example, the IP address of a component may change between two subsequent messages sent to it without being noticed by the sender.
- All connections are **outbound** which effectively means that all components may reside behind a (restrictive) firewall or a NAT-gateway. This not only increases the security of

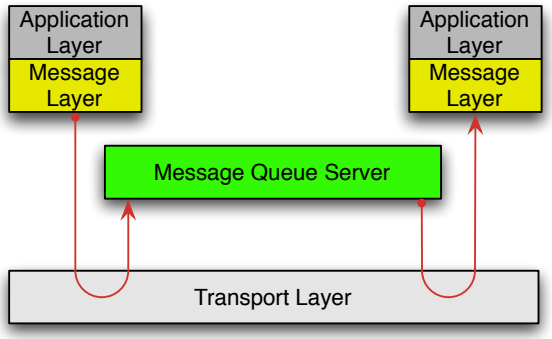


Figure 3.4.: Protocol layers in an MQS-based communication.

the *xbed*, but also targets the problems which typical network-policies and hence resulting network-layouts of grid-environments or companies impose.

- The MQS need not to be on a single machine, but can be distributed over many computers to implement **fail-over** and **load-balancing**.
- Messages can be kept in a **consistent storage** within the MQS if they cannot be delivered right now. That may happen, if the communication partner is temporarily disconnected — all pending messages will be delivered as soon as the end-point connects again.
- Multiple MQS can be configured to provide **forwarding and routing** of messages destined for a particular queue — that means independence from the actual network-topology.
- A MQS can be configured to provide **authentication** and **authorization** to limit access to particular queues.
- Messages sent from one component to another can be **transformed** while passing the MQS. That means in particular, that each component may send messages in its own native format and the MQS intelligently transforms the messages into the native format of the receiver.

The drawbacks of the usage of MQS are: An increased delay in message transmission, because all messages must be processed by the MQS before they can be delivered. And that the secure (*i.e.* encrypted) transmission of messages has to be implemented by the applications themselves. The latter issue is discussed in more detail and with regard to the XenBEE in Section 3.5.

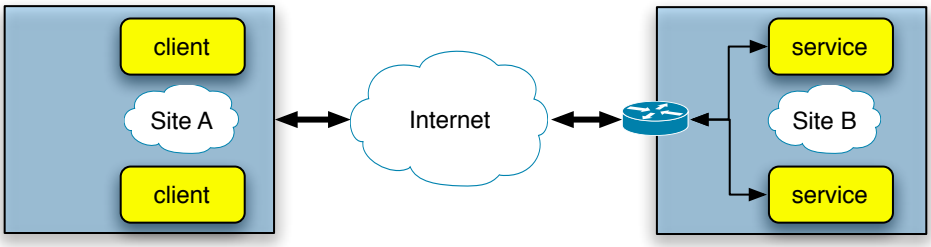


Figure 3.5.: A simple message-based system which is using a MQS.

An example for a distributed system that uses a MQS is shown in Figure 3.5. Site B has some services connected to a MQS. These services can be reached by clients from site A through an internet connection. The steps involved in building up this communication scheme are:

1. Each service connects to the MQS and *subscribes* itself to a unique queue (*e.g.* service.X).
2. Clients subscribe themselves to unique queues, too (*e.g.* client.Y).

Now that each party is subscribed to its own unique queue, a two-way communication is possible:

1. A client that wants to communicate with one of the services, sends its messages to the unique queue of that particular service. The sent message contains a special *reply-to* field that is set to the unique queue of the client
2. Answers from a service to a connected client are sent to the queue specified in the reply-to field of received messages.

The same communication scheme will be used in the XenBEE, as well. Each component subscribes itself to a unique queue, whereas the queue of the *xbed* will be configurable by an administrator.

The next section describes how the message-based communication can be secured, *i.e.* eavesdropping, modification and forgery of messages must be prevented.

3.5. Secure Communication

Since the proposed system uses message queues to transfer messages between clients and the server, all transmitted messages are sent to a message-queue server first. If this intermediate server is compromised, all messages that pass through it can also be read by the intruder.

There are two different approaches to ensure a secure transport of the messages from a client to a server and vice versa: *Transport Layer Security* (TLS, [43]) and *Message Layer Security* (MLS, [31, 63]).

Both of them use public-key certificates, *e.g.* X.509 certificates. A public-key certificate is a data structure that binds a public-key to a subject (person or system) [45]. If a communication is to be secured by the use of public-key certificates, the "users of a public-key must be confident that the associated private-key is actually owned by the correct remote subject" [45]. This can be accomplished by having a trusted authority digitally sign the involved certificates. Such an infrastructure is a called a *Public Key Infrastructure* (PKI). More information about public-key cryptography can be found in Appendix A.1 on page 77.

The following section describes the *Public Key Infrastructure* in detail. After that the *Transport Layer Security* and *Message Layer Security* protocols are discussed and analyzed. Subsequent to that the implications for the XenBEE are discussed.

3.5.1. Public Key Infrastructure (PKI)

A *Public Key Infrastructure* provides the authentication of user identities using public-key certificates. The main aspect is that there are special **trusted third parties** (*Certificate Authority*,

CA) that are permitted to **digitally sign** other certificates. If a user* wants to prove his identity to another entity, his certificate is validated by that entity against the CA's certificate. If the validity could be verified, the user successfully proved that he is in possess of the private-key that belongs to this public-key [45].

The CA is responsible for checking that the public-key contained in the certificate actually belongs to the requesting user, server or other entity denoted in the certificate. This process is for example performed by verifying the credentials of a user (*e.g.* with help of a photo identification).

Any third-party that trusts a given CA will transparently trust any entity that offers a certificate signed by that particular CA.

Validation is performed by verifying that the certificate itself has been signed by a trusted authority — the actual validation process is a little bit more complex, since it involves checking against a *Revocation List* and a "best before" date (*i.e.* life time of the certificate), too.

The signing process uses the authority's private key to compute a cryptographic signature. This private-key must of course be kept in a very secure location (*e.g.* on a physically from the Internet disconnected computer) — if it would fall into the wrong hands, the whole chain of trust is compromised.

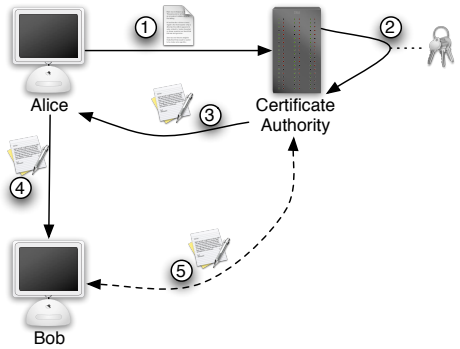


Figure 3.6.: Alice proofs her identity to Bob using a certificate that is signed by a CA that both, Alice and Bob, trust.

An example verification process is shown in Figure 3.6, the steps can be described as follows:

1. *Alice* request the signing of her certificate by a CA and thus sends a certificate request to the CA containing her public-key.
2. The CA in turn verifies Alice's credentials and eventually signs the certificate with its private key.
3. The signed certificate is sent back to *Alice* for her later use.
4. Now, *Alice* wants to prove her identity to a friend of her, *Bob*, therefore *Alice* sends her certificate to *Bob*. The proof may be necessary to establish a secure communication over an insecure channel, *e.g.* the Internet.

* or some server, etc.

5. *Bob* verifies the received certificate against the very same CA by which *Alice* had her certificate signed. Since *Bob* trusts the CA and the received certificate states, that it belongs to his friend *Alice*, he can be assured, that he is talking to *Alice*.

Both of the following protocols can make use of a PKI to authenticate communication partners. Once the communication partner is authenticated a public-key based secure communication can be set up by the protocols.

3.5.2. Transport Layer Security (TLS)

The TLS protocol (RFC 4346, [46]) can use certificates on both sides, *i.e.* client and server side. For websites server-side certificates are used so that clients can validate that they are actually communicating with the correct server. The server's certificate must therefore be signed by an authority that the user trusts. For authentication to the server client-side certificates are used. In this case the client's certificate must be signed by an authority which the server trusts.

The protocol is split into three phases [46]. Firstly the communication partners negotiate the cryptographic algorithms that should be used. Secondly certificate-based authentication and a public-key based key exchange are performed. The last phase is the actual communication phase. In this phase the transmitted data is encrypted with a symmetric encryption algorithm. The shared key that is used had been exchanged in the second phase.

Since the TLS aims to secure the transport layer (*e.g.* TCP), only direct connections between two systems are secured. When sending a message over such a connection, the whole message is encrypted prior transmitting it. On receiving a message, it is automatically decrypted.

Figure 3.7 shows the communication between a client and a server with an intermediate host. The client and the server do not have a direct connection to each other which means that all messages have to be transmitted to the intermediate host first. The individual connections to the intermediate host are secured with TLS.

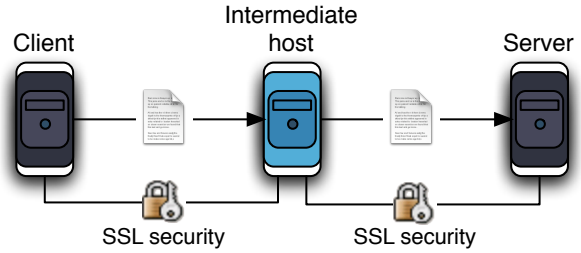


Figure 3.7.: Secure communication with Transport Layer Security (derived from [31]).

The actual communication between the client and the server is only partially secured. All transmitted messages can be accessed in their unencrypted version on the intermediate host.

In message-queue based systems there is always at least one intermediate server — the message-queue server. This means that TLS cannot be used to provide end-to-end communication security between the distributed components. But it can still be used to secure the individual connections from each component to the message-queue server.

3.5.3. Message Layer Security (MLS)

In contrast to the *Transport Layer Security* protocol the *Message Layer Security* protocol aims directly at the messages that are sent between two systems ([31, 63], MLS).

MLS is an approach that encapsulates all security related information within the transmitted message itself [31]. Securing the message instead of the transport layer has several advantages. The following list is based on the information that can be found in [31]:

- **Increased flexibility**. It is possible to secure selected parts of a message only [31]. An MQS has to inspect received messages in order to forward it to the correct destination. This part of the message can be left unencrypted while the remaining part of the message is encrypted.
- **Extensibility**. Intermediate systems or services can add their own (signed) headers to the message without breaking unrelated (*e.g.* encrypted) parts of the message. An example that is listed in [31] is *audit logging*.
- **Support for multiple protocols**. MLS can be used to send messages securely over a variety of protocols such as SMTP, FTP or TCP without relying on the security of the transport protocol [31].

The major strength of MLS is also its greatest disadvantage. Since the security information is integrated into the messages, the layout of the messages must be known to the security layer. That means, each different message layout requires an own implementation of MLS. Another disadvantage is the complexity of this protocol which imposes some overhead to the message processing step.

In [63] a specification for securing *Simple Object Access Protocol* (SOAP) messages with MLS can be found. Figure 3.8 depicts the same communication problem as Figure 3.7, *i.e.* the communication between two systems with the usage of an intermediate host.

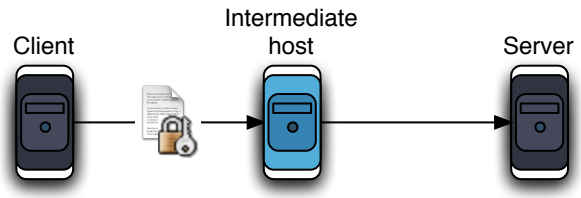


Figure 3.8.: Secure communication with Message Layer Security (based on the picture found in [31]).

The message is encrypted by the application running on the client host. In contrast to the TLS, this step is performed only once. The encrypted message is sent to the intermediate host and then forwarded to its final destination. Since strong cryptography is involved, the intermediate host cannot access (*i.e.* read) the encrypted parts of the message.

Consequently can MLS be used to provide a secure end-to-end communication for distributed applications that use message-passing.

3.5.4. Implications for the XenBEE

The previous sections have shown that only the MLS provides secure end-to-end communication for systems that use an intermediate message-queue server. Authentication, as well as strong cryptography is in both protocols provided by a *Public-Key Infrastructure*.

To ensure a secure communication between *xbe* and *xbed Message Layer Security* has to be used. To actually make sure that a client "talks" to correct server and vice versa, authentication must be provided in both directions. This implies that the XenBEE uses public-key certificates and a PKI.

"Design is not just what it looks like and feels like. Design is how it works."

(Steve Jobs)

Chapter 4.

Design and Implementation

This chapter is about the design and implementation of the *Xen-Based Execution Environment* (XenBEE). The execution environment incorporates a total of three main components: the *xbe* on the user's side, the *xbed* on the server's side and the *xbeinstd* on the side of a single virtual machine. All components have been implemented using the *Python* programming language [41]. In particular, version 2.5 of that language has been used.

Some additional modules and programs which are not part of the standard libraries shipped with Python have been used, though. Among these are for instance the `twisted` framework [58] which has been used for the network code, a library called `libvirt` [27] that was used to connect to the Xen virtual machine monitor, and a library that provides Python bindings to the `curl` library [40].

4.1. Overview

The picture in Figure 4.1 shows an overview of the three components which, when put together, make up the *Xen-Based Execution Environment* (XenBEE).

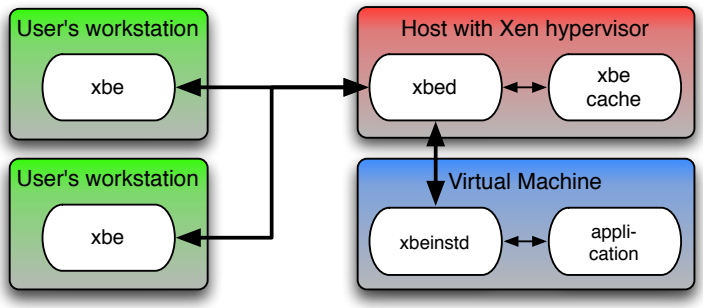


Figure 4.1.: The components of the Xen-Based Execution Environment

The "thicker" connections between *xbe* and *xbed*, as well as between *xbed* and *xbeinstd* are logical connections realized by using message-queues and one or more MQS in between. Whereas the "thinner" links are either inner-process connections (in case of the cache) or connections between parent and child process (in case of the user-application).

On the left hand side of the picture are the users using the *xbe* to communicate with the execution environment. The *xbe* refers in this case to the command line tool which I have implemented as a proof of concept to interact with the *xbed*. The interface that a user utilizes

to execute his applications with the XenBEE could be a web-portal or some other tool with a graphical user interface as well.

On the right hand side are the components that are required to execute the applications within virtual machines. The *xbed* has to be running on a machine that supports the Xen hypervisor. It maintains an internal connection to a local cache which can be used by any user to deposit arbitrary data on the server side. The *xbed* uses `libvirt` to connect to the Xen hypervisor and to manage active virtual machines.

Each virtual machine must provide the *xbeinstd*. It has to be started at some point during the initialization process of the guest operating system. Any image that is submitted to the execution environment must therefore contain this program. The *xbeinstd* is responsible for two major issues: executing the actual application and keeping the virtual machine instance alive. Execution of an application involves for instance passing arguments to the executable, setting the working directory and redirecting the input and output streams. The *xbed* is going to shut stale virtual machines down, unless the *xbeinstd* sends regular keep-alive messages to the *xbed*. If the user's application has finished, the *xbeinstd* signals the *xbed* that the virtual machine is ready to be shut down.

4.2. The Xen-Based Execution Daemon (xbed)

This section describes the "heart" of the XenBEE — the *xbed*. The *xbed* is itself composed of several smaller components that are each responsible for a single detail of the daemon. The main components though are the `TaskManager`, the `InstanceManager` and the `Cache`. The latter is a rather simple implementation of a local data cache which will be discussed in a later section.

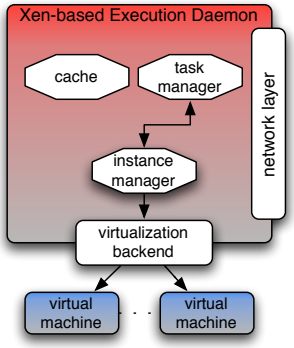


Figure 4.2.: The most important components of the xbed

Network Layer

On startup, the daemon tries to connect to the message-queue server that has been defined either in its configuration file or as a command line parameter. This process is realized by the *network layer*. It uses the `twisted` framework to establish a TCP connection. When the connection has been successfully established, a special transport protocol — the STOMP protocol [55] — is attached to the connection. STOMP is the *Streaming Text Oriented Messaging*

Protocol and basically defines a very simple protocol to send and receive text messages over a message-queue server. On top of this text message based protocol are XML based protocols that accomplish the whole communication between all components. There is currently a basic XML protocol that encapsulates single "messages", consisting just of header and body, and two protocols that connect up the *xbe* and the *xbeinst* accordingly. A special protocol providing security related services (*i.e.* privacy and validity) can be added as an additional layer.

Anyway, the details of the STOMP and XML protocols, as well as the network layer which is to some extent equal among all components of the XenBEE, will be discussed in Section 4.5, *The Communication Protocol Stack*.

Task-Manager

When a user submits, terminates or requests the status of one of her jobs, the message is actually handled by the `TaskManager`. This component controls all tasks, the system knows of. A "task" does in this case not refer to the actual application a user submitted, but to a container that holds the task's state machine, description and probably a reference to the virtual machine instance used for this task.

A new task gets initialized every time a user requests a reservation. At this time it contains nothing more than the state machine which is in its start-state (*i.e.* `Pending:Reserved`). Section 4.2.1 discusses the implementation of the job-model that has been used to represent an activity.

Each task and each reservation has its own unique identifier by which they are known to the system and to the user. Those unique identifiers are implemented by using *Universal Unique Identifiers* (UUIDs). Whereas the task identifier is more or less publicly available*, the identifier of the user's reservation is only known to that particular user (or some intermediary software such as a Calana-agent). All requests that a user makes to the system require the reservation's unique identifier.

When a user confirms a reservation, he also sends the job description as a JSDL document along with the confirmation message. The task is thus completely specified and may perform the transition into the `Executing` state. To this time, the task-manager creates a special spool directory for that task which is eventually going to contain all necessary files to create the virtual machine and execute the user's application. The details of how the required files are obtained will be discussed in a later section (Section 4.2.2). After all files have been retrieved, the `InstanceManager` component is used to create a new virtual machine for the task.

Instance-Manager

The instance-manager's purpose is to create, control, monitor and destroy active virtual machine instances. Again unique identifiers are used to name the virtual machines. To start a virtual machine several files are required:

- An operating system installation which resides in a special file-system image file.

* a listing of all current tasks comparable to the UNIX `ps` command could be possible

- A kernel and potentially an initial ramdisk image (initrd). The initrd typically contains additional device drivers and setup routines that are not directly included in the kernel.

These files must be provided by the user, because she knows best how the operating system must be set up to run the specific application.

By just using these three files, a virtual machine cannot be created right away, it must be *configured* first. The configuration of a virtual machine is manifold, it contains descriptions of the operating system that is to be used (*i.e.* the mentioned three files), memory settings (*i.e.* the amount of virtualized physical memory), the number of virtual CPUs and network parameters.

Instance Configuration and Setup The *xbed* provides the possibility to use different virtualization back-ends, but the current implementation supports only Xen virtual machines. It could, for instance, be possible to implement a back-end that uses VMWare's [60] virtual machines, even no virtual machine at all could be thinkable.

The back-end uses the `libvirt` API to connect to the Xen hypervisor. This connection is basically used to query the current state of a virtual machine and to shut a running virtual machine down. Virtual machine creation is implemented by using a call to the `xm` command line tool provided by the Xen user-space tools. Prior a new instance can be created, a configuration file has to be generated. This configuration file contains the mentioned parameters.

The network configuration of a virtual machine is based on the *Dynamic Host Configuration Protocol* (DHCP). That means, the guest OS has to be configured to use DHCP. The administrator of the host, on which the *xbed* runs, can specify a list of MAC addresses that shall be used by the virtual machines. In addition to these MAC addresses, the administrator can also specify a URI or an IP address by which the virtual machine

Virtualized physical memory and the number of virtual CPUs can be specified in the JSDL document using the predefined resource descriptions.

Instance Creation The next step is the generation of a configuration file which can be used by the back-end — in this case Xen — to set up a new virtual machine. The task's state machine is triggered to change its state to the next sub-state of `Running`: `InstanceStarting`. Once the virtual machine has been created, the *xbed* awaits a callback from the virtual machine. This callback expresses itself in form of a message sent by the *xbeinstd* running within the just created virtual machine. If this signal does not get sent within a given timeout, the virtual machine instance is assumed to be broken. The *xbed* will therefore shutdown and destroy this virtual machine. Consequently, the execution of the user's task has failed.

The callback from the *xbeinstd* fulfills two important functions. Firstly it makes sure that the virtual machine's network configuration is correct and fully functional. Secondly it is verified that the *xbeinstd* is installed in the image and did start properly. Improperly configured images are thus recognized very fast.

Now, that the virtual machine is ready to execute the user's application, the task's description may be sent to the *xbeinstd* and the task's state machine may eventually change its state to `Executing`. The details of executing the application is going to be discussed in Section 4.3.

After finishing the execution of the user's application, the virtual machine will be shut down and the result can be staged out according to the specified JSDL document.

The next sections describe the implementation of the used job-model, how the staging of input and output data is performed and how a user can benefit from using the provided data-cache.

4.2.1. Job-model Implementation

In Section 3.3 I have already discussed the usage of the job-model that has been proposed by the OGSA-BES working group. The required batch-job execution semantic of the XenBEE demands the extension of the state-model to support *data staging* states. To be integrable into existing grid-environments such as Calana the model must also contain extensions for *reservations*. Actually I am using the state-model that is currently defined for the Calana architecture as the basis.

To model the process of starting a virtual machine, I have extended the model again to provide an additional state `Instance-Starting` which is a sub-state of the basic state `Running`. The final job-model is shown in Figure 4.3.

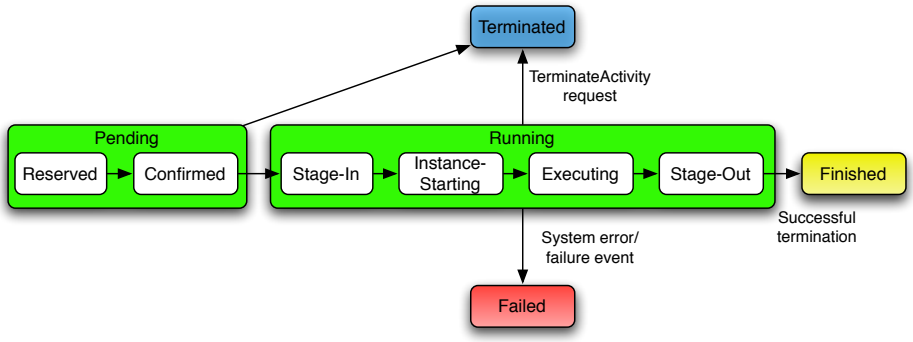


Figure 4.3.: The job-model used in the XenBEE

The implementation of this model builds up on an implementation of a *Finite State Machine* (FSM). The `Task` class contains a reference to an instance of such an FSM. Each time a state change is desired the FSM is called with an "input" signal. The FSM then calls registered functions that implement the transition's behavior. If, for example, a user wants to terminate his activity, the FSM is presented with a "terminate-token". The FSM calls then specialized functions that deal with the termination request according to the current state. That means, distinct functions are perhaps called when traversing from `Pending:Reserved` and `Running:Executing` to the `Terminated` state, respectively.

Some of the transitions involve rather complex and time-consuming actions (*e.g.* file transfers). Those complex transitions are represented by *activity-objects*. An activity-object is an object which encapsulates some behavior along with a state — commonly known as *Function*-objects or *Functors*. I named them activity-objects on purpose, because they are usually executed by a separate thread of control in concurrency to other activities within the system. Another reason for encapsulating some of the transitions in activity-objects is the possible intervention by a user.

The BES model allows a user to terminate his task at any time. In particular that means, that any action belonging to that task, which currently takes place on the server, has to be stopped or aborted.

Since the task-manager does not only create, but also manage the tasks, he is responsible for stopping current activity-objects of a task if he is asked to do so. For that reason, he is in charge of a per task list that contains all current activity-objects for that task. If the task-manager is now going to handle a request for termination of one of the tasks, he first cancels all registered activity-objects before letting the task to change its state to `Terminated`.

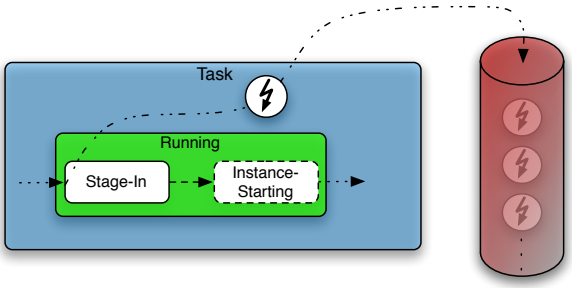


Figure 4.4.: Handling of *activity-objects* with an activity-queue.

The picture in Figure 4.4 shows such a case. The task (represented by the blue box) is currently in the `Stage-In` sub-state of `Running` and is awaiting the availability of its required files. The activity-object which represents here the operation of staging files in is shown as the encircled lightning bolt. When the task transitions into the `Stage-In` state, it registers the shown activity-object with the task-manager. The task-manager in turn adds it to his queue of current activity-objects (shown as the reddish tube in the right of the picture). When the activity-object is finished (figuratively speaking: if the lightning bolt exits through the bottom of the tube), the task may advance its state to `Instance-Starting`, *i.e.* it can be attempted to start an instance for this task.

Since the retrieval of files from different locations (specified by URIs in the JSDL) may take some time, this example also motivates the usage of threads to decouple other activities of the system from these steps. When terminating a threaded activity-object, the responsible thread is signalled to abort whatever it is doing at the time.

The whole cycle through which a task may run is depicted in Figure 4.5 on page 41 as an activity diagram. The only sub-activity that cannot be aborted at all is the *stop instance* operation. That is because the shutdown process of the underlying virtual machine just cannot be cancelled or reversed. That is also the reason to not having an extra sub-state for that operation in the job-model.

The process starts with waiting on a "ready-to-go" signal which is usually included directly in the `Confirm` message received by an user, but can also be given in a subsequent message on its own (`StartRequest`)*. The next steps resemble the previously discussed job-model. Of

* the communication protocol and the messages involved are discussed in Section 4.5, *The Communication Protocol Stack*.

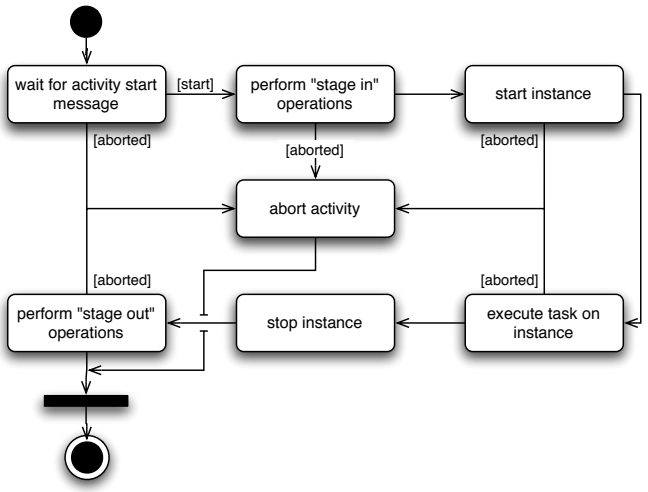


Figure 4.5.: Summary of the steps involved when executing a task.

course, any of those steps may *fail*, which effectively results in the failing of the whole task. The operations that are involved when one of the sub-activities fails are the same as for the abortion of that sub-activity. Actually, the *stop instance* operation cannot fail, since it is always possible to forcibly shut down a virtual machine.

The *start instance* operation differs from the other operations in that two components are involved, the *xbed* and the *xbeinstd*. The *xbed* first attempts to start a back-end instance (*e.g.* a Xen virtual machine) and waits for the instance to be started, eventually.

After the instance has been started, *i.e.* the back-end did not reject the provided configuration, the *xbed* (actually the thread executing this particular activity-object) waits for the *xbeinstd* to send an `InstanceAvailable` message back to the *xbed*. Figure 4.6 shows the details of that particular step.

As you can see, there are three different outcomes for this step. The instance can be marked as being available which renders the task eventually executable, the step can as well be aborted due to a request for termination by the user or the instance can fail to start at all. The task's state will be changed according to the outcome of this step.

After sending the `InstanceAlive` notification to the *xbed*, the *xbeinstd* waits for the task's description. Additionally, it will send messages to the *xbed* regularly, thus making sure, that the virtual machine is still "alive".

The next section deals with the staging operations in more detail. In particular this means, how exactly the files are retrieved, how files can be compressed to reduce the required network-bandwidth and how one can make sure that the files have been transferred correctly.

4.2.2. Data Transfer Handling

The handling of data transfer covers the staging of files into the execution environment and out from the execution environment, in this work referred to as "stage-in" and "stage-out" respectively. The description that is required for each of the staging processes is completely covered

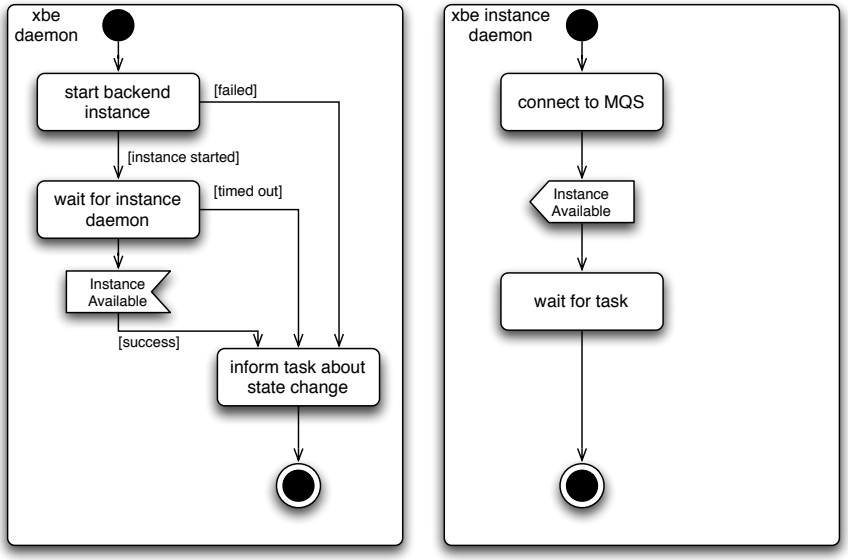


Figure 4.6.: The *xbed* waits for the virtual machine to be available. The availability of a virtual machine is made sure by waiting on a special message from the *xbeinstd*.

within the JSDL document which a user submitted to the execution environment. However, some extensions are required, those will be handled with in Section 4.2.4.

The stage-in process is twofold. The first part of stage-in is the acquirement of files that are mandatory for the execution environment to create virtual machine, these are the early mentioned *image* and *kernel* files (probably an *initrd* as well). Without these files, the next part cannot be started.

The second part of the stage-in process handles the input files that an user had specified for his application. These definitions use the standard `DataStaging` element of the JSDL specification. These files are directly retrieved into the virtual machine image which was previously obtained, hence the two parts of the staging process.

All files, including the virtual machine specific ones, are referred to as *Uniform Resource Identifier*s (URIs). That means, all files must be "somehow" accessible by the *xbed*. A user can, for instance, specify files that are located on a `HTTP` or on an `FTP` server — currently only these two protocols and a special URI to reference to cached files are supported. Those URIs are then retrieved by the *xbed* by using the standard mechanisms for file retrieval based on these protocols — the library `libcurl`, which relates to the UNIX `curl` command, is used to implement download and upload.

Support for compressed files

As said before, all input files related to the application are retrieved into the virtual machine image directly. Consequently, the image must provide enough free space to hold all input and generated output data, which can be quite a lot. Since the image is an ordinary file on a filesystem, it can easily consume unpredictable size. This image has to be transfered from the user (*i.e.* the location he specified in the JSDL) to the host on which the *xbed* runs. Fortunately, the

image is nearly empty before transmitting it, since it contains only a basic operating system installation along with the user's application and its dependencies.

The *xbed* allows a user to submit compressed files. The compression is extremely useful when the file is mostly "empty" as it is the case for the image files, for instance — an image file that was 8 GB in size and contained about 750 MB data produced a compressed file (using `bzip2`) that was about 500 MB in size only. The usage of compressed files reduces the time needed to transfer large, but mostly empty images significantly. The user is allowed to tag every file that she submits to the execution environment with the mode of compression and the *xbed* will decompress the file after retrieval.

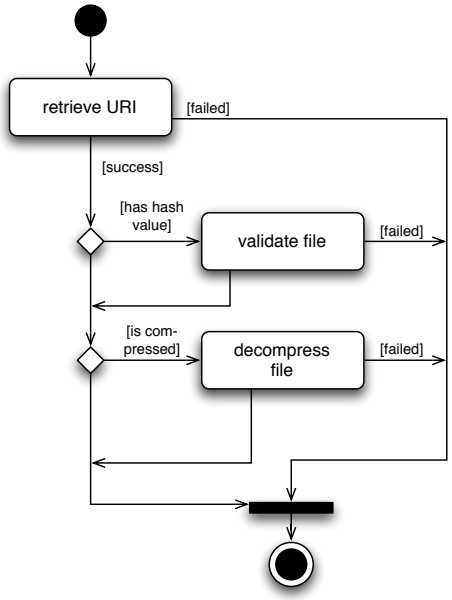


Figure 4.7.: Steps involved when retrieving a URI.

Support for validation of files

Figure 4.7 shows the involved steps when a file is retrieved by the *xbed*. Another feature added to this process is the validation of the retrieved data. If a user wants to make sure, that the file had not been modified in some way, she can provide a checksum and the used algorithm along with the URI. Typically a *cryptographic hash function* is used to compute a digital fingerprint of the data. All secure hash algorithms that are provided by the Python `hashlib` module are supported (some examples are: `SHA1`, `SHA256` or `MD5`).

The whole stage-in process

The stage-in process is split into several steps as shown in Figure 4.8. The process always starts with the creation of a `chroot` environment* within the spool directory that has been created for that task. The description of the necessary files for a virtual machine is contained in an extra

* The terms *chroot environment* and *jail environment* are used interchangeably. A short description of a `chroot` environment can be found in the glossary.

element within the task's JSDL description. A user can either specify the required files (image, kernel and initrd) on its own or he can specify a `bzip2` compressed tar archive ("package") that contains these files. Additionally, a user has the possibility to define several executable scripts that are uploaded to the execution environment and get called at various stages of the stage-in process.

After the package or the virtual machine files have been retrieved, validated and probably decompressed, the "pre-setup" hook is executed. This hook consists of the scripts that were either in the package or specified by the user and tagged to be in this hook. All scripts are executed in the previously created chroot environment so that they cannot access any file that does not belong to this task. Until now the execution environment had not touched the image file, so that one of the scripts could be used to decrypt the file, for instance.

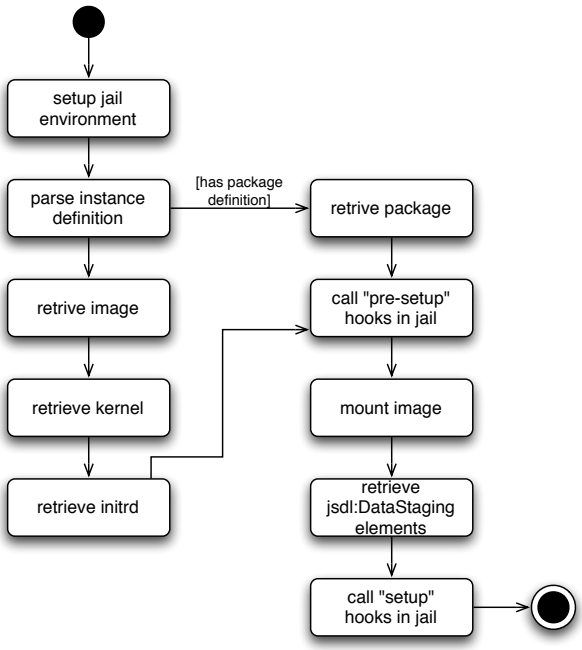


Figure 4.8.: Overview of the parts of the stage-in activity.

Now that all virtual machine specific files are available, the application specific data can be retrieved. The *xbed* assumes, that the image is mountable by the UNIX `mount` command and mounts it to a temporary location within the jail environment.

All JSDL-`DataStaging` elements that define a stage-in operation are now handled with the same retrieval mechanism as described above. The paths that are used in the job description are interpreted to be relative to the mount point of the image.

Finally the "setup"-hook is executed using the user-supplied scripts. These scripts are given the path to the image's mount-point and can, for example, decrypt already staged-in files or even retrieve additional input data. If everything went well, the virtual machine is completely set up and can be started.

Upload of files

The upload of files from the execution environment to a location specified by the user is also accomplished by using URIs. The process of uploading a file is less complicated than the retrieval process, since it currently does not involve compression or validation directly.

The first step after the virtual machine has been shut down is again the mounting of the image to a temporary location within the jail environment. To provide the possibility to compress the files before uploading them, the "cleanup" hook is called before any of the actual staging operations is called. The next step is the handling of the JSDL-`DataStaging` elements that define a stage-out operation. The only currently supported protocols that can be used for the upload are `FTP` and `HTTP`. In the final step of the upload process the "post-cleanup" hook is called; to that time, all specified staging operations have already been successfully performed and the image has been unmounted.

If everything went well, all stored data* that belongs to this task is destroyed (*i.e.* the spool directory will be deleted) and the task is put into the `Finished` state.

4.2.3. Caching Of Arbitrary Files

Compression of files can decrease the deployment time already significantly, but to avoid long-distance transfers over an unspecified network connection, the caching of files on the server side is required, this follows strictly from the *locality principle* [13]. The *xbed* supports this by providing the user with a simple data-cache incorporated into the system. The user is able to store arbitrary data on the server prior submitting a job to the system. This makes the initialization of a virtual machine for often used images a lot faster compared to always retrieving them over a potentially slow network connection.

Adding data to the cache

An user adds files to the execution environment by specifying a URI that can be retrieved by the *xbed*. This will usually be the same URI the user would have given in his job description (*e.g.* a location on some `FTP` or `HTTP` server). The *xbed* attempts to retrieve the given URI and adds a new entry to a database.

Actually the complete mechanism of caching is implemented in a special component within the *xbed* on its own. The cache uses an SQL-database to store information about the cached data in a persistent way. A user can add two different pieces of information to the data he wants to have cached. The first is the **type** of the data, which can be one of `image`, `kernel`, `initrd` or `data`, where the `data` type is just a placeholder for arbitrary data that does not fit into one of the other categories. The second piece of information is a **description** that can be used to describe the data in more detail, *e.g.* the version of a specific kernel, a list of the applications that are installed within an image or the type of compression if the data had been compressed. Additionally an `SHA1` digital fingerprint of the data is computed and stored, along with the provided information, in the database.

* this does not include internal data structures (*e.g.* exit-code).

The cache-component assigns each entry a unique identifier based on UUIDs, this identifier can later be used in a URI to refer to that entry. When using the *xbe* command line tool to add data to the cache, the URI of the newly created entry will be printed on the screen upon success.

Discovering cache entries

The first way of "discovering" an entry of the cache is simply to write down the URI one has received upon addition of the entry. But that is not always feasible, since the entry could be shared among several users who do not know each other — say, an administrator or provider of a virtual machine image added it to the cache, so that it can be used by several, to that time unknown, users.

The discovery of cached entries is implemented in the *xbed* by simply providing the user with a list of all entries. This list contains most importantly the URI of the entry, the type of data and the description the submitter had given.

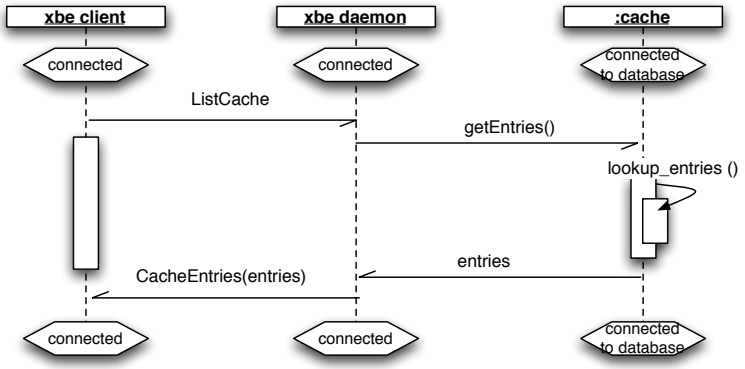


Figure 4.9.: TODO: fill me in

The *Message Sequence Chart* (MSC) in Figure 4.9 shows the messages that are sent between the client and the server. The client requests a list of all cached entries by sending the `ListCache` message. Thereupon makes the *xbed* a call to the cache-component that provides him a list of all entries. This list is eventually transformed into an XML-message and sent back to the client. A sample output of the `xbe showcache` command is shown in the following listing:

```
<CacheEntries with 1 entry:
{'cache://xbe-file-cache/adfd0a99-4761-4816-8902-db8d62c8e482':
  {'description': 'Ubuntu kernel (2.6)',
   'hash': '732ca16f330c2c382e83cb997f5e027687a285aa',
   'type': 'kernel'}}
>
```

Listing 4.1: *xbe* output of cache entries.

Using cache entries

Cache entries can be used rather easy. The user is just required to specify the URI of a cache entry instead of the original URI. The *xbed* will lookup the specified URI during the stage-in process and retrieve the data from the cache, if a valid entry could be found. The same retrieval mechanism applies to the cached entries, so that compression and validation can be used with them, too.

4.2.4. The Xen-based Submission Description Language (XSDL)

This section describes the extensions to the *Job Submission Description Language* that were developed. The extensions have been required, since the JSDL does not directly support the description of virtual machines.

There are actually two different kinds of extensions: general extensions, that can also be used to enhance some of the standard JSDL elements, as well as an extension that is purely specific to the proposed execution environment.

General extension elements

As described above, the XenBEE supports the submission of compressed files and the validation of those files based on digital fingerprints. To reflect this behavior along with the job submission, additional "decorator" elements have been added. The following example (Listing 4.3) shows the usage of these decorators. The JSDL does only support plain URIs. If such a URI is now used along with one or more of these decorators, the *xbed* interprets the URI correctly.

```
<jsdl:Source>
  <jsdl:URI>
    ftp://ftp.example.com/pub/input-file
  </jsdl:URI>
  <xsdl:Hash algorithm="sha1">
    a3b180e5dc2359849ffa927b93414ada20807a0c
  </xsdl:Hash>
  <xsdl:Compression algorithm="bzip2"/>
</jsdl:Source>
```

Listing 4.2: Example with the general `Hash` and `Compression` extensions.

The example in Listing 4.2* could be an excerpt of a JSDL-`DataStaging` operation. The `Source` element contains the URI that refers to the location of an input file which should be staged in.

The `Hash` extension stores the digital fingerprint of the source file and the algorithm that has been used to generate it. This fingerprint is used by the *xbed* to verify that the retrieved file is actually the same as on the server. The `Compression` extension holds the algorithm only —

* The namespace prefixes refer to the namespaces defined in *The Extensible Markup Language*, Table 3.1.

one of `bzip2`, `gzip`, `tbz` and `tgz` — that has been used to compress the file. In this case, the file `input-file` is compressed with the `bzip2` algorithm.

An XML-Schema document is used to validate each one of the extensions. For example, the compression algorithm is checked against the list of supported algorithms and the content of the `Hash` element is restricted to hexadecimal digits.

When the document is parsed a special class will be instantiated for each of these elements. That are the `Compression` and `HashValue` classes. Both are initialized with the parameters given to the element and are used to decompress or validate the retrieved file respectively.

XenBEE specific extensions

The staging operations that are provided by the JSDL cannot be used to describe the required files for a virtual machine (*i.e. image*, *kernel* and *initrd*). Those operations "work" on the file-system hierarchy that is perceived by the application itself, and that is already the file-system of the virtual machine.

However, the virtual machine specific files have to be staged in as well, so there was a need to provide an extension to the JSDL. The extension follows the JSDL in the way it is structured and is added as an additional element to the JSDL-`Resources` element. The extension defines an `InstanceDefinition` element which contains the actual `InstanceDescription`. The following listing shows a thorough example usage of this extension.

```
<xsdl:InstanceDefinition>
  <xsdl:InstanceDescription>
    <xsdl:Instance>
      <xsdl:Image fs-type="ext3">
        <xsdl:Location>
          <xsdl:URI>http://www.example.com/base.img</xsdl:URI>
        </xsdl:Location>
      </xsdl:Image>
      <xsdl:Kernel>
        <xsdl:Location>
          <xsdl:URI>http://www.example.com/kernel</xsdl:URI>
        </xsdl:Location>
      </xsdl:Kernel>
      <xsdl:Initrd> <!-- the initrd is optional -->
        <xsdl:Location>
          <xsdl:URI>http://www.example.com/initrd</xsdl:URI>
        </xsdl:Location>
      </xsdl:Initrd>
    </xsdl:Instance>
  </xsdl:InstanceDescription>
</xsdl:InstanceDefinition>
```

Listing 4.3: Example of an `InstanceDefinition` element describing the required files of a virtual machine.

Each one of the special files has its own element, *i.e.* the `Image`, `Kernel` and `Initrd` elements, whereas the `Initrd` element is optional. The locations of these files are specified by using

an instance of the `Location` element that contains a URI. The image specification can contain an additional attribute that defines the used file-system — currently only `ext2` and `ext3` are supported.* The generic extensions discussed before can be used with the `Location` elements as well.

The here defined files are staged directly into the spool directory that has been created exclusively for this single job, whereas the JSDL-`DataStaging` files are retrieved into the image.

4.3. The Xen-Based Execution Instance Daemon (xbeinstd)

The *Xen-Based Execution Instance Daemon* is a rather simple component which runs within a virtual machine created by the *xbed*. The user or the person who provides the image for a virtual machine is required to install this daemon inside the image prior submission. The daemon must also be started during the initialization of the operating system (an `init`-script to start and stop the daemon on UNIX-like systems is provided in the source package).

When the *xbed* creates a new virtual machine it adds two parameters to the kernel which are eventually exported to the `init`-script as environment variables. The first parameter contains the unique identifier of the instance, so that a the *xbeinstd* is aware of the virtual machine identification allocated by the *xbed*, whereas the second contains a URI. The URI† specifies the message-queue server and the queue that shall be used to contact the *xbed*. Both parameters together are used to establish a two-way communication between *xbed* and *xbeinstd* based on message-queues.

Startup of the xbeinstd

On startup, the *xbeinstd* uses the URI given in the environment variable `XBE_SERVER` or as a command line parameter to connect to the message-queue server. When the connection could successfully be established, the *xbeinstd* subscribes itself to a unique queue which uses the identifier of the virtual machine it is running on.

Now that the *xbeinstd* is completely set up, it notifies "his" *xbed* that it is available now. Therefore it sends a simple `InstanceAvailable` message to the *xbed* and waits for a job that it can execute. Additionally to that, it sends regular `InstanceAlive` message to the *xbed*. These messages tell the *xbed* that the instance is still functional‡, they also contain some informational values such as the uptime and idle-time of the virtual machine.

Execution of the user's application

Once the *xbed* is aware of the availability of the virtual machine and the *xbeinstd*, it sends the job description using the `ExecuteTask` message to the virtual machine. The *xbeinstd*, in turn, parses the JSDL document contained in this message.

The JSDL supports the description of executables on POSIX-compliant systems using the `POSIX-Application` extension. The individual steps that lead eventually to the execution of the

* Both of them are standard file-systems found on Linux systems.

† The URI could, for example, look like: `stomp://mqs.example.com/xenbee.daemon.1`

‡ The *xbed* shuts virtual machines down, if they do not send keep-alive messages regularly.

application can be summarized as follows:

- The `Executable`, `Argument` and `Environment` elements are parsed. These values are assembled to the parameters that are eventually passed to the `execve` system-call.
- The input, output and error streams of the application are redirected either to the files specified in the JSDL document, or to `/dev/null`.
- The working directory of the application is set either to the directory specified in the JSDL document, or to "`/`".
- Finally the application is executed as a child process of the *xbeinstd*.

Additionally to the environment variables specified in the job description, an "`MQS`" environment variable is set. Per default, this variable is set to the message-queue server that is also used by the *xbeinstd*, but the value can be overridden by a user using an `Environment` element that explicitly defines this variable. An application that is distributed over several virtual machines can make use of this variable for communication and coordination purposes.

When the application finishes its execution, the *xbeinstd* notifies the *xbed* about that using an `ExecutionFinished` message. The daemon does not terminate itself, it rather waits for the *xbed* to shut the virtual machine down. I decided to implement it in that way, to have the option to possibly reuse the virtual machine, *i.e.* in the future, it could be possible to execute more than one application in the same virtual machine.

On-demand virtual machine deployment

In the previous section I have described how ordinary applications are executed with the *xbeinstd*, but those applications typically terminate after some time. The on-demand deployment of a virtual machine does not aim at the execution of an application specified in the JSDL, it just aims at the creation of the virtual machine.

To keep the *xbeinstd* simple, the best way was to define a special extension to the JSDL. This extension aims at the `Application` element just as the POSIX extensions. The element is called `XBEApplication`. The following listing shows the usage of this extension. The only "executable" that is currently defined is the `ContinuousTask`.

```
<jsdl:Application>
  <xbe:XBEApplication>
    <xbe:ContinuousTask/>
  </xbe:XBEApplication>
</jsdl:Application>
```

Listing 4.4: Example of a `ContinuousTask` used for on-demand virtual machine deployment.

When the *xbeinstd* encounters this specific `Application` element, it does nothing, actually. The user must log in to the created virtual machine, *e.g.* by using a remote shell such as SSH.

4.4. The Xen-Based Execution Command Line Client (xbe)

The implemented command line client is very straightforward to use. It provides the user with an on-line help system, which provides information about all supported commands and how they are used. The help system can be activated by issuing the `xbe help` shell command in a terminal.

The *xbe* has an interface to all aspects of the execution environment that have been discussed so far. It allows a user to `reserve`, `confirm` or `terminate` a reservation and it is also possible to monitor the `status` of a given reservation. The tool too has an interface to the cache and it is able add data to the cache and list the current contents by using the commands `cache` and `showcache`, respectively.

Since the protocol assumes a two-step process for the submission of a new job (*i.e.* creation and confirmation of a reservation), the *xbe* provides a convenience command, that allows a user to perform the two steps in one `submit` command. The `submit`, as well as the `confirm` commands require the job definition as a JSDL document. The creation of such a document is not part of the XenBEE, but several example files are included in the source distribution.

The next section is going to discuss the implemented message-based communication protocol in detail.

4.5. The Communication Protocol Stack

The three components that have been discussed in the previous sections are using message-queues and one or more message-queue servers to communicate with each other.

This section describes the details of the communication architecture and its individual layers. There are four layers, the *Application Layer*, that implements the behavior of each single message, the *XML Message Layer* that represents each message as an XML document, the *Security Layer* which provides authenticity and privacy as described in Section 3.5, *Secure Communication*, and at the lowest level the *STOMP Layer* which is used to communicate with a message-queue server. An overview over the layers is shown in Figure 4.10 on page 52.

In the *Application Layer* a message is represented by an object, so that information contained in the message can easily be read and written. The message is then transformed into an XML-document and passed to the next layer, *i.e.* the *XML Message Layer*. The *Security Layer* is actually an XML-based layer, too, it signs and encrypts the message received from the upper layer in an XML-message. The final step is the sending of the message to the communication partner's message-queue using the STOMP protocol.

The following sections describe the common format of the XML messages that are used in the security layer, as well as in the message layer. After that each layer of the protocol stack is discussed in detail, starting with the STOMP layer. The application layer is not described here, since it has already been covered in the previous sections.

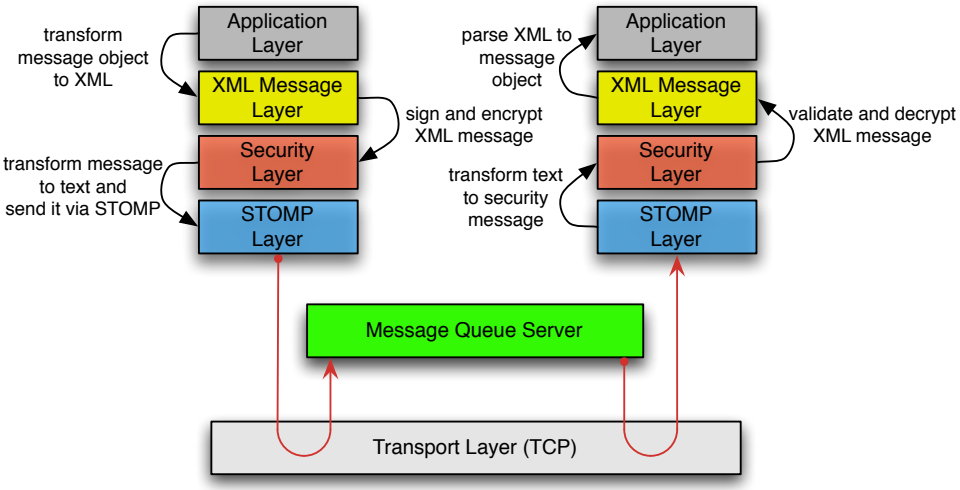


Figure 4.10.: The individual protocol layers that are involved when sending a message to another component.

4.5.1. XML Messages

The XML messages that are used in the *Security Layer* and the *XML Message Layer* share a common structure. The messages are split into two parts *MessageHeader* and *MessageBody* which are both children of the top-level *Message* element.

The header, for example, may contain security related information, currently it is only used by the security layer. The body, however, can either be empty, too, or contain another XML document that represents the application data.

```
<xbe:Message>
    <xbe:MessageHeader>
      <!-- security related information, etc. -->
    </xbe:MessageHeader>
    <xbe:MessageBody>
      <!-- payload XML document representing
           application data. For example:
           an encrypted XML document, Error, etc. -->
    </xbe:MessageBody>
</xbe:Message>
```

Listing 4.5: The structure of an XML message.

The `Error` message is used in all XML-based protocol layers of the XenBEE to indicate error or "status" conditions. This message may be sent in reply to any other message except another `Error` message. The following table (Table 4.1) lists the attributes of such an error message.

Some important error codes, their names and a short description are listed in Table 4.2. The error code 200 is actually not an error in the common sense, it is a "no-error" error (it is comparable to the HTTP Status Code 200).

attribute	description
code	The error code
description	Generic information about the error
message	A descriptive message of the exact error that occurred.

Table 4.1.: Attributes of the `Error` message.

To transfer an XML document, the internal tree representation is converted into a textual representation and then sent to the communication partner through the lower transportation layer.

code	name	description
200	`OK`	everything is okay
300	`SERVER_BUSY`	the request could not be handled right now
400	`ILLEGAL_REQUEST`	the received message was invalid, *e.g.* XML validation failed
501	`TICKET_INVALID`	the client specified an illegal reservation identification (ticket)
502	`SECURITY_ERROR`	the message-layer security could not be established
503	`UNAUTHORIZED`	the user's certificate is not allowed

Table 4.2.: Important error codes and their descriptions.

4.5.2. The STOMP Layer

The *Streaming Text Oriented Message Protocol* (STOMP) [55] is the layer at the lowest level and is used to communicate with a message-queue server.

When one of XenBEE's components wants to communicate with another component, it connects to a message-queue server using TCP. This connection can now used to establish a Stomp-connection. If a user wants to request the status of one of his submissions, he passes a URI to the *xbe*. This URI contains the low-level transport mechanism, the address of the message-queue server and the queue that must be used to communicate with the other side. Consider the following URI:

```
stomp://mqs.example.com:61613/a
```

It specifies, that Stomp should be used to connect to the message-queue server that is listening at the given address. To reach the service on the other side, messages are sent to the queue `"a"` on this message-queue server.

Since the available Stomp clients for Python did not meet my expectations and requirements — they did not integrate well with the `twisted` framework, I implemented the client side of the

protocol myself. Therefore, the source distribution of XenBEE includes a generic implementation of the Stomp protocol that can be used in other projects as well. Additionally, a small command line tool (`stompclient`) has been implemented which can be used for testing purposes — it supports the subscription to multiple message-queues, as well as the transmission of files to an arbitrary destination queue.

Protocol Description

The Stomp protocol comes "from the HTTP school of design" and the client is really easy to implement. The site at [55] states, that even a `telnet` session can be used to communicate with a Stomp server.

The messages that are sent between client and server are called *Frames* and consist of three parts: *Command*, *Header* and *Body*. The *Command* defines the type of the frame (*e.g.* `SUBSCRIBE` is used to subscribe to a message-queue on the server, `SEND` is used to send an application message). The *Header* consists of key-value pair lines; the header's end is denoted by an empty line. The *Body* contains the payload of the frame and is ended by a `null` (control-@ in ASCII) byte. This section does only describe those frames that are relevant to the implementation of the XenBEE.

Connecting to a Stomp Server

After a client is connected to a Stomp-server, *e.g.* by using a TCP-connection, it has to login before it can actually use the connection to send and receive messages — this is achieved by sending a `CONNECT` frame to the server as shown in the following Listing 4.6.

```
CONNECT
login: <username>
passcode: <passcode>

^@
```

Listing 4.6: The initial `CONNECT` message sent by a Stomp client.

The `CONNECT` frame supports two header elements, `username` and `passcode` that can be used by the server administrator to restrict the access to the server. The server sends either an `ERROR` frame which contains detailed information about the occurred error or it sends the `CONNECTED` frame (see Listing 4.7).

```
CONNECTED
session-id: <session-id>

^@
```

Listing 4.7: The `CONNECTED` message sent by a Stomp server after a client has successfully logged in.

According to the Stomp protocol specification found in [55], the `session-id` header element is a unique identifier for this session, but it is not actually used yet.

Subscribing to Message-Queues

A Stomp client may subscribe to as many queues or topics as the application requires (see Listing 4.8 for an example frame). The Stomp server takes control of the subscriptions and forwards all messages that are targeted at a particular queue or topic. *Topics* and *queues* differ mainly in the way messages are handled. A topic has the semantics of a $m : n$ communication*, *i.e.* a message that is sent to a topic will be received by **all** subscribers. *Queues*, on the other hand, have $m : 1$ semantics — a sent message is only received by **at most one** client that is subscribed to that queue. If more than one client did subscribe to the same queue, the server may randomly select one of them to be the receiver.

```
SUBSCRIBE
destination: /queue/foobar
ack: client

^@
```

Listing 4.8: The `SUBSCRIBE` frame used for queue or topic subscription.

The frame that is shown above subscribes the client to the queue `foobar` and will receive messages targeted to that queue. Stomp distinguishes queues and topics by special prefixes in the `destination` header element, *i.e.* `/queue` and `/topic`, respectively. The header element `ack` can take on two different values `client` and `auto`. If it is set to `client`, the recipient of the message must send an `ACK` frame to acknowledge the reception, otherwise the frame is not considered to be delivered and will be sent again later. If it is set to `auto`, the frame will be sent only once† and is discarded afterwards.

To end a previously made subscription, the client can send an `UNSUBSCRIBE` frame (see Listing 4.9). The following listing cancels the subscription to the `foobar` queue:

```
UNSUBSCRIBE
destination: /queue/foobar

^@
```

Listing 4.9: The `UNSUBSCRIBE` frame revokes a previously made subscription.

The components of the XenBEE use queues to establish their communication. The *xbed* subscribes to a queue that looks like `xenbee.daemon.<unique-id>`, however, the actual queue has to be configured by an administrator (*i.e.* specify the unique identification). The *xbeinstd* subscribes to queues of the form `xenbee.instance.<UUID>` — the UUID is the unique identifier of the virtual machine instance. The *xbe* is no exception to that and subscribes to queues of the form `xenbee.client.<UUID>`, where the UUID is randomly chosen.

* m is the number of clients that send messages to that topic or queue, n is the number of subscribers.

† A message-queue server delivers messages not until at least one client is subscribed to that queue.

Sending and Receiving Messages

To send application data to another entity, the sender uses a `SEND` frame. An example of such a frame is shown in Listing 4.10. The shown frame represents the transmission of the message "Hello World!" (without the quotation marks) to queue `a`. The `reply-to` header is used to tell the receiving entity to which queue it has to address its replies.

```
SEND
destination: /queue/a
reply-to: /queue/b
content-length: 12

Hello World!^@
```

Listing 4.10: The `SEND` frame is used to send application data.

The `content-length` header defines the actual length of the payload (the empty line between header and body marks the end of the header). The string "Hello World!" consists of exactly 12 characters. The Stomp client can now be sure that the next `null` byte indeed marks the end of this frame. If no `content-length` is specified, the Stomp client assumes that the frame ends at the first occurrence of a `null` byte. Depending on the payload data, this can be harmful — consider, for example, the transmission of raw, *i.e.* non-ASCII, data, in this type of data a `null` byte is very likely to occur.

This message is transmitted by the server to one of the consumers that are subscribed to the queue "`a`". The following listing (Listing 4.11) shows how the resulting frame could probably look like:

```
MESSAGE
destination: /queue/a
message-id: <message-identifier>
reply-to: /queue/b
content-length: 12

Hello World!^@
```

Listing 4.11: The `MESSAGE` frame is used by the server to transmit a message to a client.

The header elements `destination`, `reply-to` and `content-length` are still the same. But a new header element has been added by the server, too. The `message-id` header uniquely identifies this message — if the queue-subscription was set to `client`-acknowledge mode, this identifier has to be used in the subsequent `ACK` frame.

This concludes the basic transportation layer that is used in the XenBEE. All messages that are received by the Stomp layer are forwarded to the upper layer and will be handled by specialized protocols.

4.5.3. The Security Layer

This layer implements the *Message Layer Security* functionality as described in Section 3.5.3. The protocol is based on XML messages that are sent through the lower layer. Currently, the *Security Layer* is only used in the communication between *xbe* and *xbed*, *i.e.* between user and server.

The protocol is split into two phases, an *initialization phase* and the actual *communication phase*. During the initialization phase the communication partners exchange their public-key certificates and set up the encryption scheme that shall be used in the next phase. The certificates are then validated against a Certificate Authority to verify the identity of the communication partner. The subsequent communication phase uses the exchanged information of the first phase and transports application data in a secure way (*i.e.* digitally signed and encrypted).

Initialization Phase

This phase is entered whenever one of the endpoints wants to start a new communication with another endpoint. That means both sides can establish a new connection, but most of the time it will be the client side (*e.g.* the *xbe*) that initiates the connection. Before the actual communication can be secured, the public-key certificates of both endpoints must be exchanged.

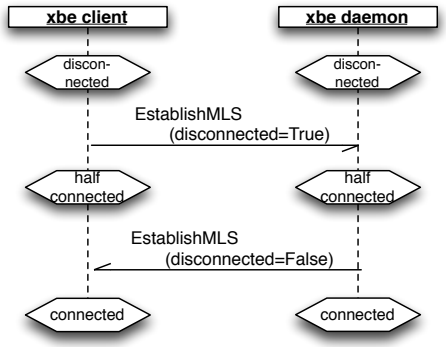


Figure 4.11.: Exchange of public-key certificates.

The MSC in Figure 4.11 shows which messages are sent to establish a secure communication. They start both in the `disconnected` state and the client starts the initialization process by sending a `EstablishMLS` message to the server. This message contains the client's certificate and the signature in the header of the message. The message also says that the client is currently in the `disconnected` state which tells the server to send his certificate as well.

The server receives the message, verifies its signature using the certificate included in the message and finally validates this certificate against a CA. If the certificate is valid, the server checks whether the certificate owner is authorized to connect at all. If the certificate could not be validated (*i.e.* the signature did not match or the certificate was not issued by the CA) or the owner of the certificate is disallowed to connect, an `Error` message indicating the reason for failure is sent back. If the user is allowed to connect and the received message has been valid, the server sends a `EstablishMLS`, too. As requested by the client, this message includes the server's certificate, but this time the `disconnected` flag is set to `false`.

All transmitted messages are digitally signed by the sender. At the end of the two-way handshake, both sides know the other's certificate, so that a public-key based, secure communication can be established.

Communication Phase

This phase handles the signing and encryption of the actual application data (*payload messages*). The signature of the payload is added directly to the `MessageHeader` of the payload, while the encryption of a message creates a new message, the *envelope message*. The schematic process of transmitting a message from the *xbe* to the *xbed* is shown in Figure 4.12.

Before a message is encrypted, it is signed by the sender, the procedure for generating the signature can be summarized in the following steps:

- Generate the canonical representation of the *payload-message's* XML document (see [66] for more information on *Canonical XML*).
- Transform this canonical form into a textual representation and compute its digital fingerprint using a secure hash algorithm.
- Sign this digital fingerprint with the sender's private key and add the resulting signature to the header of the payload message.

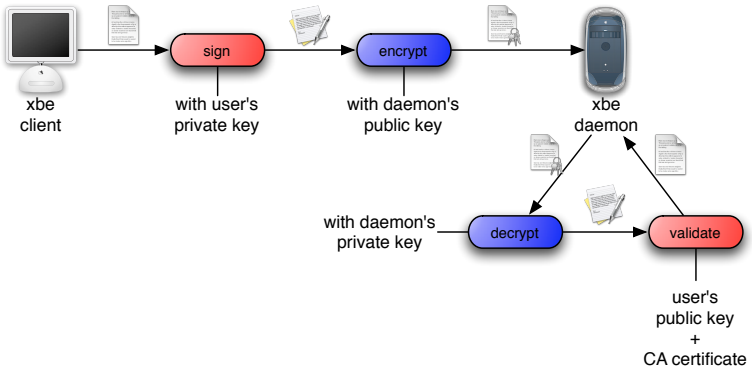


Figure 4.12.: Secure transmission of an XML document.

To validate such a message, the recipient must remove the signature from the message's header first. Now he can compute the digital fingerprint of the message in the same way as the sender. Finally, to actually validate the message, the signature is validated with the sender's public-key — if the result is equal to the fingerprint, the message is valid.

The next step is the encryption of the signed message, therefore the just mentioned envelope message is created. The encryption is performed by using a symmetric encryption algorithm with a randomly generated key*. This key is then itself encrypted using the recipient's public-key and added to the header of the envelope message. The body of the envelope consists of the encrypted message in a `base64` encoding. The skeleton of such an envelope message is shown in Listing 4.12.

* The current implementation uses the `DES-EDE3-CBC` algorithm. Detailed information on this algorithm can be found in RFC 2420, *The PPP Triple-DES Encryption Protocol* and in [52].

```
<xbe:Message>
  <xbe:MessageHeader>
    <xbe-sec:CipherInfo>
      <xbe-sec:CipherKey>encrypted key</xbe-sec:CipherKey>
      <xbe-sec:CipherIV>Initialization Vector</xbe-sec:CipherIV>
      <xbe-sec:CipherAlgorithm>
        encryption/decryption algorithm
      </xbe-sec:CipherAlgorithm>
    </xbe-sec:CipherInfo>
  </xbe:MessageHeader>
  <xbe:MessageBody>
    <xbe-sec:CipherData>
      <xbe-sec:CipherValue>encrypted message</xbe-sec:CipherValue>
    </xbe-sec:CipherData>
  </xbe:MessageBody>
</xbe:Message>
```

Listing 4.12: The skeleton of an encrypted message.

The first implementation of all security related functions made use of the `M2Crypto` library [28]. Unfortunately, during the tests of the XenBEE on a 64-bit architecture this library failed to work, even though it worked on a 32-bit architecture just fine. I decided to drop this library and implement the same functionality using direct calls to the `openssl` executable. This is just an workaround until the problems with the `M2Crypto` library are solved.

The next sections describe the *XML Message Layer*. There are actually two different protocols involved, one for the communication between the user (*xbe*) and the *xbed* and another one for the communication between a virtual machine (*xbeinstd*) and the *xbed*.

4.5.4. Communication between xbe and xbed

This section describes the messages that are sent between a client application, *i.e.* the *xbe*, and the *xbed*. All messages have the same, basic structure, *i.e.* `MessageHeader` and `MessageBody`, whereas the body contains the actual content. Prior any application data is exchanged, the *Message Layer Security* is established and I assume in the subsequent sequence diagrams, that both *xbe* and *xbed* are already in the `connected` state as shown in Figure 4.11.

Create a new reservation

To create a new reservation with the *xbed*, the client sends a `ReservationRequest` message. The server checks, if it can accept a new reservation currently (current load, number of virtual machines, etc.) and returns a `ReservationResponse` message or an `Error` indicating that the request failed. If the reservation was successful, the `TaskManager` is used to create a new task object. A successful reservation is shown in Figure 4.13.

The `ReservationRequest` contains no attributes to the time of the implementation, since a special resource managing and description system was out of the scope of this work.

The returned `ReservationResponse` message has only one attribute: the `ticket` which uniquely identifies the just made reservation. This ticket is only known to the client and the

server and must be used by the client in any subsequent request that concerns this reservation.

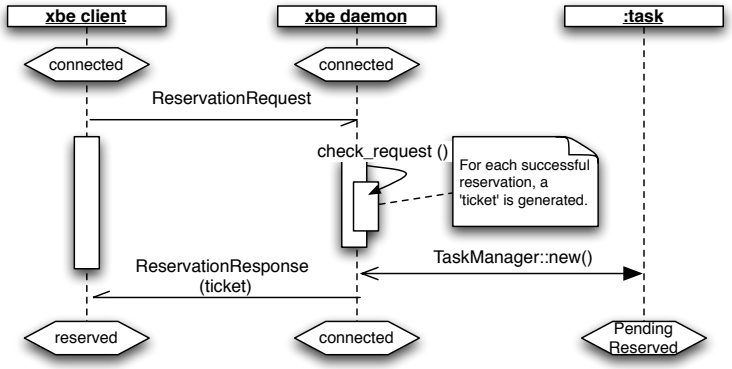


Figure 4.13.: Create a new reservation with the *xbed*.

Confirm a reservation

The next a user may want to do, is to confirm the reservation — as shown in Figure 4.14. Therefore, the client sends a `ConfirmReservation` message. On reception of this message, the server first validates the contained JSDL document against the JSDL specification, then it is checked if an `InstanceDefinition` element is included and finally the validity of this description is checked as well. If any one of these checks fails, an `Error` is returned and the user has the possibility to modify the submission and resend the `ConfirmReservation` message. If the document is completely valid, the task is looked up and the transition from `Pending:Reserved` to `Pending:Confirmed` is performed.

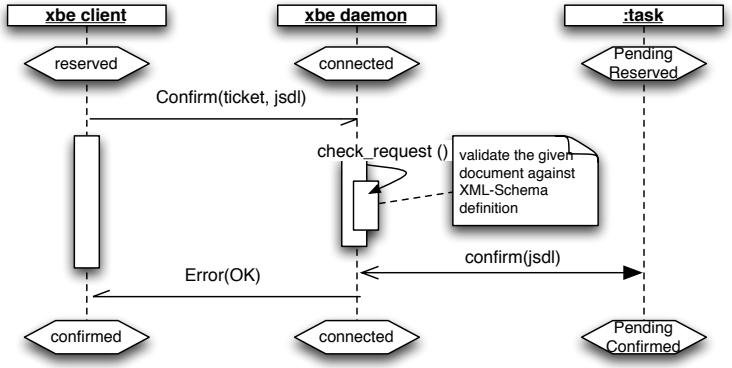


Figure 4.14.: The *xbe* confirms a previously made reservation.

The `ConfirmReservation` message contains the previously obtained ticket, the JSDL document and a `start_flag` (an overview can be found in Table 4.3). If this flag is set, the *xbed* performs the transition into the `Running` state automatically as soon as possible. If the flag is not set, the task remains in the `Pending:Confirmed` state until the client sends an explicit `StartRequest` message.

attribute	description
ticket	the reservation identification number
jsdl	the JSDL-document that describes the task
start_task	a flag that indicates whether to start the task immediately

Table 4.3.: Attributes of the `ConfirmReservation` message.

Request the status of a task

To request the status of a task, the client sends the `StatusRequest` message to the *xbed* (Figure 4.15). This message contains the ticket that is required to refer to the reservation. The server looks up the task that belongs to the given reservation and retrieves the current status of the task.

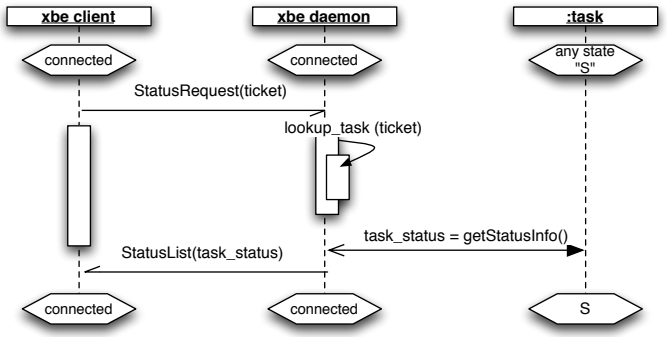


Figure 4.15.: TODO: fill me in

The reply from the server is a `StatusList` message that contains the current state (using the XML representation proposed by the BES specification [33]) and other information about the task. Table 4.4 shows a summary of the included attributes. The exit code is only available, if the task is already in the `Finished` state.

attribute	description
entries	list of `Status` elements, whereas each entry represents a single task and contains the following attributes
task-id	the unique identification of the task
state	the current state (BES specification format)
meta	meta information such as log messages, exit code, various timestamps

Table 4.4.: Attributes of the `StatusList` message.

The `meta` information included in the `StatusList` message is a dictionary (*i.e.* a list of key-

value pairs), that contains information that could be useful for a human being or possibly the client. There are, for example, timestamps that are set when a transition is completed. If the virtual machine is already running, the current IP-address is included, too.

Terminating a reservation

If a user wants to terminate his reservation, the client simply sends a `TerminateRequest` that contains the identification number for the reservation along with a reason why the termination is requested. In addition to that, the user can tell the *xbed* to remove the data structures that would else be kept for later reference.

attribute	description
ticket	the reservation identification number
reason	the reason why the reservation shall be terminated
remove_entry	a flag that indicates to automatically remove the entry after it is terminated

Table 4.5.: Attributes of the `TerminateRequest` message.

The server replies to this message with an `Error` message which either indicates that the termination is in progress or that some error occurred.

Messages related to the Cache

The next messages are those that currently provide the access to the server-side cache. A user can add a new entry to the cache by sending a `CacheFile` message that contains the required information, *i.e.* the URI of the data that shall be cached, a description and the type of data.

To request a list of all currently cached files, the user has to send a `ListCache` message. The reply to this message is a `CacheEntries` message that contains a description for all entries. The attributes of this message are shown in Table 4.6

attribute	description
entries	list of `Entry` items with the following attributes
uri	the URI of this cache entry
description	description of the entry
type	type of cached data
hash	the digital fingerprint of the data

Table 4.6.: Attributes of the `CacheEntries` message.

To remove a file from the cache, the user has to send a `CacheRemove` message which contains the URI of the cache entry.

4.5.5. Communication between xbeinstd and xbed

The communication between the instance daemon and the *xbed* is currently provided by just five different messages. The `InstanceAvailable` message is the first message that is sent be the *xbeinstd* after it has been started. The continuously sent `InstanceAlive` message contains runtime information about the virtual machine, such as the uptime and the length of time for which the system has been idle.

The `ExecuteTask` message is used by the *xbed* to send a job description to the virtual machine and the `ExecutionFinished` is sent back when the job finishes, it just contains the exit code of the application. When a user request the termination of her reservation, the *xbed* sends a `TerminateTask` message to the *xbeinstd*. This message leads to a clean and correct shutdown of the application.

"There are three kinds of lies: lies, damn lies, and statistics"

(Benjamin Disraeli)

Chapter 5.

Results

This chapter provides you with the results of the experiments I carried out. Before I am going to examine the performed experiments in detail, I will shortly describe the experimental setup which has been used. All experiments have been accomplished with the usage of two separate, dedicated machines, which means no other user could execute programs on the machines. Both machines were connected to each other by a 100 Mbit/s Fast Ethernet link that provided an average throughput of 11.78 MByte/s. The link was not shared with other machines, so that the transfer rate was not disturbed by other network activities.

In Figure 5.1 the setup of the machines is shown. The *xbed* is running on the machine that provides the Xen hypervisor. A message-queue server* has been installed on that machine as well. This message-queue server was used by all components of the XenBEE in each performed experiment.

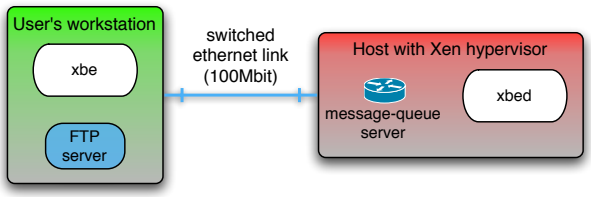


Figure 5.1.: The network setup that has been used in the experiments.

The only required program on the workstation of the user was the *xbe* command line tool. To keep the setup simple, I also installed an FTP server on this workstation machine. This FTP server delivered required files for the experiments (*i.e.* virtual machine images, input files, etc.) and it has also been used for the upload of generated output files.

Basically, both machines are of the same type, but the Xen-host system was running in 32-bit mode, while the other one used 64-bit. More detailed information on the hardware and operating system of both systems can be found in Table 5.1.

The next sections discuss the experiments I have performed to support my assumptions about the execution environment, *i.e.* that it is actually able to execute user-provided applications. Each experiment has been executed five times and only the average values of the measured times are shown in the experiment's results. The time measuring has been performed by the

* I used the *Apache ActiveMQ* server (`http://activemq.apache.org/`) for the message brokering.

		Xen-host	
	Workstation	Domain-0	Host
Kernel	`2.6.19-4-generic-amd64`	`2.6.19-4-server`	
CPU type	AMD Athlon™	AMD Athlon™	
Clock frequency	2009 MHz	2009 MHz	
CPU count	2	1	2
Main memory	2048 MByte	512 MByte	2048 MByte

Table 5.1.: Machine configurations of the user's workstation and the Xen-host.

job-model implementation — each time a new state is entered, the current time is remembered. These times are then included in the status messages as special meta information.

The first part shows some example executions such as a very simple *"Hello World!"* execution or a more advanced execution which deploys a website and provides accessibility to the virtual machine by using an SSH server.

The final part of this chapter analyzes the overall performance of the XenBEE. In particular, the influence on the total execution time when using cached or compressed images is studied.

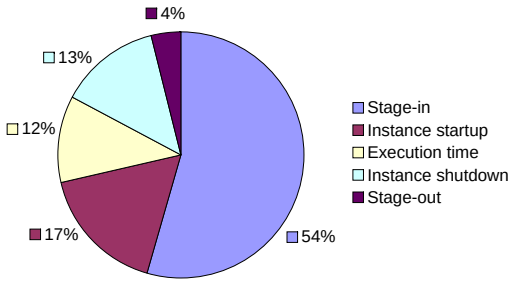
5.1. Execution examples

This section shows you three example executions where each one of them targets a different execution scenario. All experiments in this section use the same virtual machine configuration: 1 virtual CPU, 128 MByte virtual main memory and 256 MByte swap space. The operating system that was used for the virtual machines is a stripped-down Ubuntu installation (450 MByte) with a Xen-aware Linux kernel (version `2.6.19`, about 8 MByte in size).

5.1.1. Hello World!

The first execution example is very simple — it just executes the `echo` command with parameters that I supplied in a virtual machine. The generated output is then transfered back to my workstation. This example is my contribution to the list of implementations that output the string "`Hello World!`".

Figure 5.2 shows the total execution time of the job broken down into the times that were spent in each step of the execution. Since the runtime of the program itself is very short — about $0.013\ s$ on the workstation — the overhead introduced by the execution environment makes up the largest part of the total execution time. Most of the time was spent for transferring the image and for starting up and shutting down the virtual machine. The image can be transferred in about $38\ s$ from the workstation to the Xen-host — the remaining $8\ s$ of the Stage-in step are spent on setting up the jail environment and the swap space. During the Stage-out step the output of the program is transfered back to the user's workstation so that I could have a look at it.



Execution step	Time (s)
Wait for start message	0.38
Stage-in	46.14
Instance startup	14.19
Execution time	9.78
Instance shutdown	11.31
Stage-out	3.21
total	85.01

Figure 5.2.: Distribution of the total execution time of the `Hello World!` example over the individual steps.

5.1.2. Complex Computation

The previous example was rather a proof of concept execution than an actually applicable execution. A user would not want to execute a job that simple and short with the overhead of a complete remote virtual machine. The following example however could be an imaginable real world problem.

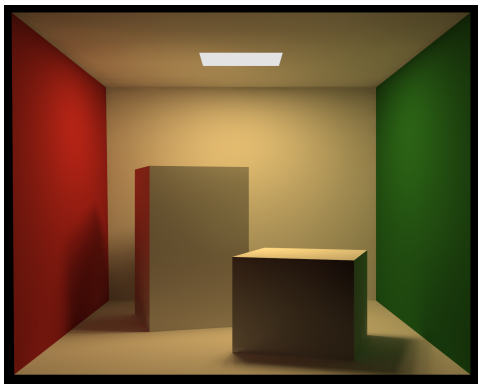
In this case the user wants to render various scenes into pictures by using the POV-Ray [39] application. This example relates to the example used in Section 2.1.2, Batch job execution. Listing 5.1 shows the important parts of the job description. The first thing to note is the usage of logical file system names (the `SPOOL` for example). The first argument to the `povray` executable is the scene definition and the last argument is the name of the output file. The stage-in step loads the `scene.pov` file into the `SPOOL` directory while the stage-out step takes the generated `scene.png` file and transfers it back to the user.

The resulting image* is shown in Figure 5.3. The virtual machine image that has been used for this example was the same as for the previous example, thus the times needed to stage in the files in both cases are nearly the same.

The execution makes this time use of stage-in and stage-out operations. In the stage-in step, a scene description is loaded into the execution environment. This description is then passed to the `povray` executable along with additional command line parameters (*e.g.* to define the rendering quality or the required size of the output picture). The stage-out step is again used to retrieve the generated output from the execution environment.

In comparison to the *Hello World*-example where the execution time was barely a fourth of the stage-in time, the execution time outweighs the stage-in time this time with a factor of 12. If the actual computation time is just long enough, the overhead that is introduced by the virtual

* The command line to produce it was: `povray +Q11 +A0.01 +W1600 +H1280 +Ocornell.png cornell.pov`



Execution step	Time (s)
Wait for start message	0.50
Stage-in	47.04
Instance startup	15.41
Execution time	569.81
Instance shutdown	11.39
Stage-out	3.36
total	647.51

Figure 5.3.: The `cornell` scene that is included in the POV-Ray program's examples — executed with the XenBEE.

machine preparation gets more and more insignificant. The same execution takes about 406 s when executed directly on the workstation, thus the pure execution of the program is still much faster.

```
<jsdl-posix:POSIXApplication>
  <jsdl-posix:Executable>/usr/bin/povray</jsdl-posix:Executable>
  <jsdl-posix:Argument filesystemName="SPOOL">
    scene.pov
  </jsdl-posix:Argument>
  <!-- quality settings and output redirection omitted -->
  <jsdl-posix:Argument>+Oscene.png</jsdl-posix:Argument>
  <jsdl-posix:WorkingDirectory filesystemName="SPOOL"/>
</jsdl-posix:POSIXApplication> <!-- ... -->
<jsdl:DataStaging>
  <jsdl:FileName>scene.pov</jsdl:FileName>
  <jsdl:Source>
    <jsdl:URI><!-- location of scene file --></jsdl:URI>
  </jsdl:Source>
</jsdl:DataStaging>
<jsdl:DataStaging>
  <jsdl:FileName>scene.png</jsdl:FileName>
  <jsdl:Target>
    <jsdl:URI><!-- destination of result picture--></jsdl:URI>
  </jsdl:Target>
</jsdl:DataStaging>
```

Listing 5.1: Important parts from the JSDL to describe a `povray` execution (Note: the listing does not describe a valid JSDL document).

5.1.3. Deployment of a Web Server

This example shows how a small web server can be deployed using the XenBEE — it relates to the example used in Section 2.1.3, Batch job execution. The virtual machine image can be generic, *i.e.* it needs only to contain the web server application and must make sure that the

server gets started during the initialization process.

To make this example more generic, the content of the web server is provided as a stage-in item as well. The content can for instance be staged in by transferring a compressed `tar` archive to the execution environment. Listing 5.2 shows again the important parts of the job description that has been used.

```
<jsdl:Application>
  <xbe:XBEApplication>
    <xbe:ContinuousTask/>
  </xbe:XBEApplication>
</jsdl:Application>
<jsdl:Resources>
  <jsdl:FileSystem name="HOME">
    <jsdl:MountPoint>/root</jsdl:MountPoint>
  </jsdl:FileSystem>
  <jsdl:FileSystem name="WWW">
    <jsdl:MountPoint>/var/www</jsdl:MountPoint>
  </jsdl:FileSystem>
</jsdl:Resources>
<jsdl:DataStaging>
  <jsdl:FileName>.ssh/authorized_keys</jsdl:FileName>
  <jsdl:FilesystemName>HOME</jsdl:FilesystemName>
  <jsdl:Source>
    <jsdl:URI><!-- authorized_keys location --></jsdl:URI>
    <xsdl:Mode>0600</xsdl:Mode>
  </jsdl:Source>
</jsdl:DataStaging>
<jsdl:DataStaging>
  <jsdl:FileName>site.tar.bz2</jsdl:FileName>
  <jsdl:FilesystemName>WWW</jsdl:FilesystemName>
  <jsdl:Source>
    <jsdl:URI><!-- location of content archive --></jsdl:URI>
    <xsdl:Compression algorithm="tbz"/>
  </jsdl:Source>
</jsdl:DataStaging>
```

Listing 5.2: File system definitions for a web server deployment.

Since the job defines the deployment of a server and not a batch job, the application is set to the special `ContinousTask` provided by the XenBEE. The description defines two logical file systems, the `WWW` directory and the `HOME` directory. The `WWW` directory is used to specify the location where the web server expects its contents and the `HOME` directory is used for the SSH login that I will describe in a moment.

The complete content (*i.e.* directories and files) for the web server is stored in a compressed `tar` archive. This file is staged into the `WWW` directory and the *xbed* decompresses it at this specific location — note the usage of the compression algorithm.

To be able to change the content or to modify the web server's configuration after the virtual machine has been started, I installed an SSH server into the image as well. The SSH server was configured to allow access only by public-key authorization. Thus there was an additional

staging operation required which loads an `authorized_keys` file into the virtual machine. Due to security reasons, the SSH server requires that this file can only be accessed by the user himself — the special `Mode` element takes care of that.

I submitted the job to the *xbed* and after about $60\ s$ the web server was running. The address of the virtual machine is included in the status message, so I could point my web browser to the address and browse through the deployed content.

5.2. Performance Analysis

The previous section showed that a considerable part of the total execution time is spent in the Stage-in step. This section will analyze this step in more detail. Two different approaches that could improve the execution time are discussed as well.

The bare execution time of a job cannot be decreased by the execution environment as such, because it mainly depends on the implementation of the virtualization back-end and the hardware that is being used. The only step in the execution path that can effectively be modified and improved is the stage-in process.

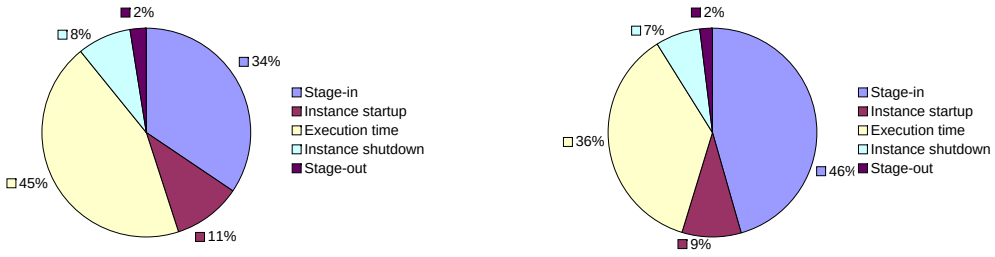
The current implementation of the XenBEE offers two options to improve that part, *compression* and *caching* of user supplied files. The following sections deal with those two possibilities.

5.2.1. Compressed Images

The experimental setup uses the POV-Ray raytracer to render a picture, but this time a less time consuming configuration* has been used. The same image as in the previous examples is used. The uncompressed image has a size of 449 MByte that can be transferred from the workstation to the Xen-host in about $41\ s$ (assuming an average throughput of 11 MByte$/s$). The 449 MByte image can be compressed with the `bzip2` program† to a 145 MByte sized file which can be transferred in about $13\ s$. Figure **??** shows the distribution of the total execution time on the individual steps.

* The execution takes about $60\ s$ on a virtual machine and only $21\ s$ on the workstation.

† I applied the `-3` option to `bzip2` to get a faster compression/decompression behavior.



(a) Uncompressed image (450 MByte) (b) Compressed image (145 MByte)

Figure 5.4.: Comparison of uncompressed/compressed images (small image).

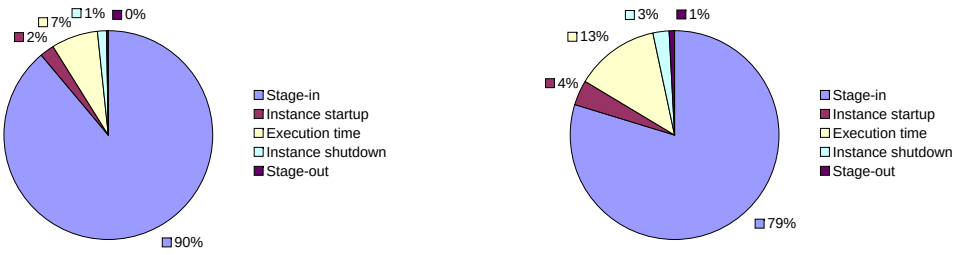
Surprisingly, the total execution time of the job had not decreased, as I thought. The execution time had even increased by circa 30 s. Table 5.2 shows the exact times that have been measured, it also shows that the main divergence occurs during the Stage-in process. The transfer of the compressed image can be accomplished in about 12 s, but the complete Stage-in step consumed 74.55 s. That means, circa 60 s have been spent for the decompression of the image. The decompression alone consumes more time than the transfer of the uncompressed image which took only 46.24 s.

	Time (s)	
Execution step	uncompressed	compressed
Wait for start message	0.40	0.40
Stage-in	46.24	74.55
Instance startup	14.20	14.94
Execution time	59.65	59.68
Instance shutdown	11.29	11.29
Stage-out	3.21	3.25
total	134.99	164.11

Table 5.2.: Comparison of the execution times when using uncompressed or compressed images, 450 MByte and 145 MByte respectively.

This example used an image that was mostly filled with data. A typical usage scenario for the XenBEE would however be to use a sparse image, *i.e.* an image that is mostly empty. The following execution example uses an image with 8 GByte of total size but only about 1 GByte is actually used. Compressing this image results in a file that is 499 MByte large. Both files can be transferred from the workstation to the Xen-host in circa 745 s and 45 s respectively. In order to be faster, the decompression has a time buffer of 700 s.

This time the usage of a compressed image outperforms the uncompressed variant signifi-



(a) Uncompressed image (8 GByte) (b) Compressed image (499 MByte)

Figure 5.5.: Comparison of uncompressed/compressed images (large image).

cantly. The time consumption distribution on the individual steps of both variants is shown in Figure **??**. Due to the difference of the total execution times, the diagram does not clearly visualize the improvement. The exact timings that were used to generate diagram are shown in Table 5.3. Staging in the compressed image is twice as fast as staging in the uncompressed image and the overall execution time dropped to circa 55% of the original execution time.

	Time (s)	
Execution step	uncompressed	compressed
Wait for start message	0.45	0.50
Stage-in	729.72	359.18
Instance startup	17.60	17.70
Execution time	58.49	58.51
Instance shutdown	11.36	11.41
Stage-out	3.38	3.41
total	821.00	450.71

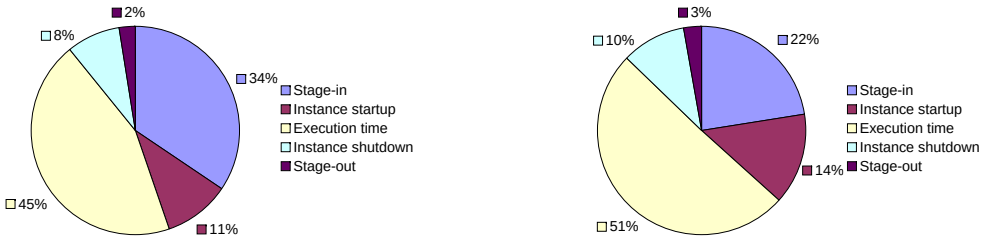
Table 5.3.: Comparison of uncompressed/compressed images (large image).

5.2.2. Data Caching

The caching of data can always be used but the gain in execution performance is greatest when the cached data is used fairly often. A virtual machine image that contains one or more applications that will be used by several users for many job executions should be cached on the server side, since the transfer rate is boosted nearly to the maximal reachable rate.

Currently, the cache is implemented by storing the data on the Xen-host. Using a cached file therefore means, that it has to be copied into the task's spool directory. The upper bound of the transfer rate for the network connection, as well as for the internal cache is thus limited by the writing performance of the file system that is used on the Xen-host. I determined this

performance on the Xen-host with a benchmark tool called `bonnie++`*. The determined pure-write performance on the Xen-host was about 40 MByte/s which is nearly four times faster than the average network throughput. Figure 5.6 shows the gain of execution performance when using a cached image for the same job execution as for the non-caching variant.



(a) POV-Ray example without cache (b) POV-Ray example with cached image

Figure 5.6.: Comparison of POV-Ray job with and without cache usage.

The stage-in of the cached image should have taken about 16 s provided that the file is written in 10 s and eventually set up after another 6 s. The additional 6 s are due the jail environment setup. But the actual time required to stage in the virtual machine files was only reduced by about 45% (see Table 5.4 for the exact values). This may be explained by the fact that the cached file had to be read and written from the same physical device.

	Time (s)	
Execution step	no cache	cached image
Wait for start message	0.42	0.41
Stage-in	46.22	25.68
Instance startup	14.14	16.24
Execution time	59.79	58.15
Instance shutdown	11.28	11.30
Stage-out	3.20	3.22
total	135.05	115.00

Table 5.4.: Comparison of the execution step times of the POV-Ray example without and with using a cached image.

* The homepage of the `bonnie++` benchmark tool can be found at `http://www.coker.com.au/bonnie++/`

5.2.3. Conclusions

The shown example executions always made use of rather large virtual machine image, but it could also be thinkable to use very small images to provide "instant virtual machines".

Since the *xbeinstd* must be installed in the virtual machine image, a complete Python installation has to be installed as well. The Python libraries, as well as the required `twisted` framework need about 25 MByte. With some work an image could be created that needs less than 100 MByte and still can contain an application that does not require large input files. Virtual machines using such images could be deployed in about half a minute. Unfortunately, I could not come up with such an image in time, so that I cannot provide time measurements.

My conclusions of the performance analysis is to use caching as often as it is possible and compression only when sparse files are used. The caching of an image does not make sense if the image is used only once, because it still has to be transferred to the Xen-host prior it can be used. But if the image will be reused over and over again, it should be cached.

Compressed images can always be used if the transfer time has to be reduced at all costs. This may involve that the complete execution takes longer than the ordinary execution with a raw image. On the other hand, if the image is sparse, compression can provide a significant speed-up of the execution time.

Both techniques can be used simultaneously to get the best of both worlds (*i.e.* caching of compressed images).

Chapter 6.

Conclusions and Future Work

This chapter analyzes the work that has been done in respect of the initial goals that have been set and provides some hints for future developments.

Conclusions

This work designed and implemented an execution environment that is based on virtual machines. The XenBEE supports two different types of job execution: a batch-job semantic and an on-demand server deployment semantic.

In addition to other products such as Amazon EC2 [16] and The XenoServer Open Platform [26], this work adds batch-job execution capabilities to virtual machine environments.

To execute such a batch-job, the user submits a virtual machine image that contains the application he wants to execute. The XenBEE creates a new virtual machine that uses this supplied image and executes the specified application within the virtual machine. Along with the submission, the user can also specify stage-in and stage-out operations that transfer data into the virtual machine or from the virtual machine to the user. Currently only HTTP and FTP are supported for these operations but the XenBEE can easily be extended to support more protocols.

As the results in Chapter 5 show, the implementation works and performs well.

With the use of virtual machine images a common problem of grid environments has been solved: It is not always clear if and where an application is installed on a given target system. This problem has been avoided by letting the user actually provide the target environment. Additionally, a virtual machine is never shared so that security for the job execution is provided.

The other execution semantic is on-demand server deployment. In this case the user supplies a virtual machine image that contains a server application such as a game server or a web server. The XenBEE creates a new virtual machine and the server is deployed. Files can be staged-in and staged-out as well, *e.g.* to provide initial content for web server.

The results have shown that the deployment of a web server works and took about $60\ s$.

The communication model that is used in the XenBEE is message-based and supported by a message-queue server. The use of a message-queue server makes it possible that the client can be behind a NAT gateway (all connections are outbound).

To provide authentication, authorization and secure communication, the XenBEE uses Public Key Infrastructure (PKI). That means each client and the server is in possess of a certificate. These certificates are signed by a trusted Certificate Authority which means client and server can verify each other's identity. For authorization the server uses a list of authorized certificates.

To provide a secure communication between client and server Message Layer Security is used. This prevents eavesdropping, tampering and message forgery, *i.e.* confidential data can be transmitted between client and server.

Security is currently only assured between client and server but neither for data transfers, nor between the server and a virtual machine. Both can be implemented easily. For the data transfer, the client would transfer credentials over the secured connection to the server. For a secure communication between the server and a virtual machine a pre-shared key can be chosen by the server prior virtual machine creation.

The XenBEE supports the use of a local cache and file compression/decompression. This can be used to speed up the virtual machine creation, because the image need not to be transfered over a (slow) network connection. In Chapter 5 the benefits of caching files has been shown.

Future work

The current implementation of the XenBEE is already usable for real world problems as it has been shown in the previous chapter. But there are still many aspects that could be implemented and analyzed in the future. The following sections provide a few ideas for future works.

Integration into Calana

The most crucial future development step is the actual integration into an existing grid environment. The execution environment understands a commonly used language for the job submission and a formalized job-model. The basic requirements for the integration into Calana have been implemented, as well. But the glue between the XenBEE and Calana (or some other grid middleware) — the Calana-agent — is still missing.

Support for Workflows

The current implementation supports only a very basic workflow, *i.e.* stage-in of input data, job execution and stage-out of generated data. A sophisticated work flow description language supports constructs for looping and conditional branches.

The support workflows could be implemented either on top of the XenBEE, or directly into the XenBEE. An independent "on top" approach only uses the execution semantics already provided by the XenBEE. This means it must always wait for the results of an execution to be staged out before the next step of the work flow can be submitted. This approach can make use of existing technologies.

An optimized implementation that is integrated into the XenBEE could prepare the next virtual machine while the current execution is still running. The results of the current execution could then be directly staged into the new virtual machine.

Enhanced up- and download mechanisms

The current implementation of the XenBEE does only support the HTTP and FTP protocols to perform upload and download operations. Other mechanism such as SCP, rsync or GridFTP should be supported as well. The access to the different storage areas could be granted to the *xbed* by user-supplied certificates.

Cache hierarchies

A cluster of machines that are used for the XenBEE could use one or more shared caches. If a user wants to execute a particular job many times or on several machines at the same time, he could load the image into the shared cache first. Each involved execution host could then retrieve the image into its local cache.

Advanced file system support

The current implementation makes the assumption that only a single image file is involved. This image contains all required data. But it could also be possible to provide access to network file systems such as NFS or GridFS and so on.

Generation of virtual machine images

The images that were used in the performed experiments have been created by a toolchain that was included in the Ubuntu distribution (`xen-tools`). These tools provide everything that is needed to create a virtual machine image, but the created images are rather large. To create really small images one has to modify the created image by hand afterwards.

User-friendly front-ends

The *xbe* command line tool has mainly been implemented as a proof of concept and to be able to actually execute example jobs. Currently the *xbe* does only support the submission of already existing job description documents.

A graphical user interface could show a list of available images to a user. In the case of on-demand server deployment a user could simply double-click on one of the available servers to start it. This can also be coupled with the support for work flows, *i.e.* the user models the work flow graphically and submits it to the XenBEE.

Appendix A.

Additional Background Information

This chapter provides some additional background information that may be useful to fully understand particular design and implementation aspects.

A.1. Public-key cryptography

Public-key cryptography describes a form of cryptography where a user holds two different keys, a *private key* and a *public key*. These two keys are mathematically related to each other, but nobody can practically derive the private key from the public key.

The public key can be made publicly available without any risk, while the private key must be kept very secret. A widely used algorithm is the RSA algorithm named after its creators Rivest, Shamir, and Adelman [47]. It has been the first algorithm, that was suitable for both encryption/decryption and signing. For more background information on public-key cryptosystems, you are encouraged to read [14, 47].

The RSA algorithm relies on the fact, that the factorization of reasonably large numbers is computationally very hard and no efficient algorithm is publicly known. Especially hard to factor are numbers whose factors are two randomly-chosen prime numbers of sufficient length.

In the following I am going to describe how the keys are set up and how they are used to encrypt/decrypt or to sign/validate a clear text message. The provided material is based on the information found in [47] and [14].

According to [47] places each user his encryption procedure E in a publicly accessible file (*e.g.* database). Using this public file, any other user is able to retrieve the encryption procedure of some other user (*i.e.* the one he wants to send encrypted messages). Each user keeps his decryption procedure D secret.

The mentioned procedures D and E have the following properties:

(a) Deciphering the enciphered form of a message M yields M. That is,

$$D(E(M)) = M. \tag{A.1}$$

(b) Both D and E are easy to compute.

(c) The user does not reveal an easy way to compute D if he makes E publicly available.

(d) The enciphering of a previously ciphered message M results in M. That is,

$$E(D(M)) = M. \tag{A.2}$$

A function E satisfying (a)–(c) is said to be a *"trap-door one-way function"* and if it also satisfies (d) it is a *"trap-door one-way permutation"* [14, 47].

A.1.1. Key setup

The *encryption key* consists of a pair of positive integers (e, n), where e is the *encryption exponent* and n is used for modulo operations. The *decryption key* is also a pair of two integers, where only the exponent differs, thus (d, n) is the decryption key and d represents the *decryption exponent*. (e, n) are made publicly available.

Rivest, Shamir, and Adelman [47] suggest the following approach for the generation of (e, n) and (d, n). The fist step is to compute n as the product of two very large, "random" primes p and q:

$$n = p \cdot q.$$

Although you publish n, nobody is able to compute the factors p and q in reasonably time due to the enormous difficulty of factoring n. In [47] it is assumed, that the computation of p and q from a given n takes 1.5×10^{29} operations, given that n has a length of 300 digits. If one operation took one microsecond, the whole computation takes 4.9×10^{15} years.

The next step is to choose d, therefore one picks a large, random integer that is *co-prime** to $(p-1) \cdot (q-1)$.† In other words, d has to satisfy:

$$gcd(d, (p-1) \cdot (q-1)) = 1.$$

Finally, the integer e is computed from p, q and d to be the *multiplicative inverse* of d, modulo $\phi(n)$:

$$e \cdot d \equiv 1 \ \ (mod\ (p-1) \cdot (q-1))$$

A.1.2. Encryption and Decryption

If two persons, Alice and Bob, want to send each other private (*i.e.* encrypted) messages, they both retrieve the other's publicly available encryption key first — Bob retrieves (e_a, n_a) and Alice retrieves (e_b, n_b).

Let's say Alice wants to send a private message to Bob. To encrypt the message, she has to represent it as an integer between 0 and $n_b - 1$ (long messages can be broken into smaller blocks, so that each block fulfills the requirement). Alice then encrypts the message M by raising it to

* two integers a and b are said to be co-prime if they do not have a common factor other than 1 and -1, *i.e.* their greatest common divisor (gcd) is 1.

† the term $(p-1) \cdot (q-1)$ is the result of *Euler's Phi* or *Euler's totient function* (ϕ) applied to n

the e_bth power modulo n_b, the result is the cyphertext C:

$$C \equiv E(M) \equiv M^{e_b} \pmod{n_b}.$$

On reception of the cyphertext C, Bob raises it to the d_bth power modulo n_b. He is the only person, who knows d_b and therefore he is solely able to decrypt C:

$$M \equiv D(C) \equiv C^{d_b} \pmod{n_b}.$$

A.1.3. Signing and Validating

Electronic signatures, *e.g.* in electronic mail systems, especially when used in business transactions, must provide provability to the receiver that the message originated from the sender. This is more than just provide mere *authentication*, where the recipient of a digitally signed message can verify that the message came from the sender. Digital signatures must be able to be used to convince a "judge", that neither the recipient did forge the message, nor the sender can deny sending the message.

That means, an electronic signature must be *message*-dependent, as well as *signer*-dependent. If the signature did not depend on the message itself, a dishonorable recipient could just change the message or attach the signature to a completely different message before showing the message/signature pair to a judge. If the signature would not depend on the *signer*, obviously anybody could have signed the message.

If Bob want to send Alice a signed message, he applies his *decryption* function D_b to the clear text message M, which results in the signature S:

$$S = D_b(M).$$

To perform this, the cryptosystem has to be implemented with *trap-door one-way permutations*, *i.e.* property (d) must hold.

This signature can now be encrypted using Alice's public key to ensure privacy, there is no need to send the message along with the signature, since it can be computed from it. On reception, Alice first decrypts the message which results in the plain signature S again. Applying Bob's *encryption* function to the received signature (Alice knows who the presumed sender of the message is) makes perfect sense due to property (d):

$$M = E_b(S)$$

Bob cannot later on deny that he sent the message, since nobody but him could have generated the signature S. Alice is able to convince a "judge", that Bob did send the message, since $E_b(S) = M$. But Alice cannot modify M or provide a different message M' because then she would also need to compute $S' = D_b(M')$ as well.

A.2. Calana

Calana is a new Grid scheduler approach proposed by M. Dalheimer [10]. The scheduler uses several *agents* and at least one *broker*.

The agents are responsible for single resource. That means they know whether the resource is free or not. They are also capable to acquire new or cancel previously made reservations on this resource.

If a user wants to submit a job to the grid environment, the broker initiates an *auction* among the connected agents. The job is then assigned to the resource that belongs to the agent that won the auction.

A.2.1. Architecture

An abstract view over the architecture of Calana is shown in Figure A.1. For a detailed discussion of the protocol that is used to perform the auctions, have a look at [9] and [35].

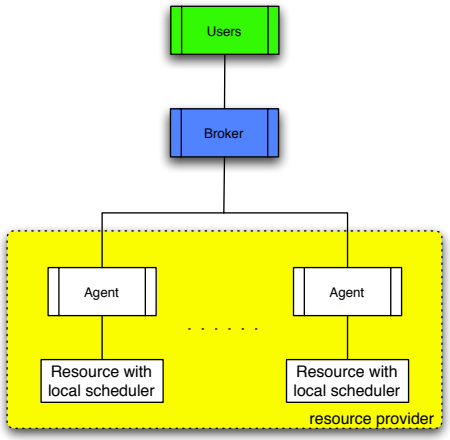


Figure A.1.: Architecture of Calana

The main steps of such an auction are can be described as follows:

1. When a user submits a job to the Calana-broker, the broker will open up an auction and try to *book* a resource for the task.
2. For each task an auction is created by sending `BookingReq`-messages to the connected agents.
3. The agents will make one or more *reservations* on their local scheduler and answer with a `AuctionBid`. Bids contain for example the cost of using the resource and various reservation parameters such as the earliest start-time and duration of the reservation.
4. To make a decision, the broker judges all received bids and chooses the best one according to some preference-model [10, 35].
5. If the user accepts the decision, the broker *confirms* the reservation.

A.2.2. Job-state model

To reflect the possibility to *make*, *confirm*, *use* and *cancel* reservations on some resource, the job-state model had to be extended. We discussed about a common state-model for the jobs and came to the consensus of adopting the BES model to our needs (see Figure A.2).

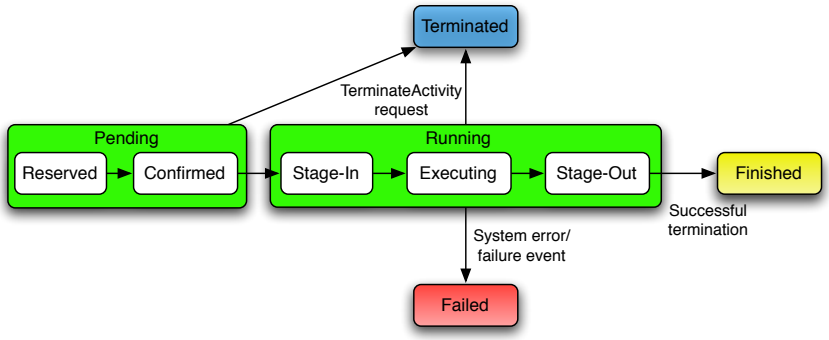


Figure A.2.: The common job model proposed by the Calana Grid scheduler.

References

[1] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[2] bochs. The Open Source IA-32 Emulation Project (Home Page). URL `http://bochs.sourceforge.net/`. Date of last visit: 14th February, 2007.

[3] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Syst. J.*, 28(1):104–123, 1989. ISSN 0018-8670.

[4] J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the AFIPS National Computer Conference 1973*, 1973.

[5] A. Church. An Unsolvable Problem Of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[6] Condor. Condor Project Homepage. URL `http://www.cs.wisc.edu/condor/`. Date of last visit: 21st April, 2007.

[7] CORBA. The OMG's CORBA Website. URL `http://www.corba.org/`. Date of last visit: 22nd April, 2007.

[8] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[9] M. Dalheimer. Calana Protocol Definition. Working Paper, 2006.

[10] M. Dalheimer, F. Pfreund, and P. Merz. Agent-based Grid Scheduling with Calana, 2005.

[11] DCOM. COM: Component Object Model Technologies. URL `http://www.microsoft.com/com/default.mspx`. Date of last visit: 20th April, 2007.

[12] Peter J. Denning. Performance modeling: Experimental computer science at its best. *Communications of ACM*, 1981.

[13] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005. ISSN 0001-0782.

[14] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[15] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[16] Amazon EC2. Amazon Elastic Compute Cloud, Virtual Grid Computing: Amazon Web Services. URL `http://www.amazon.com/gp/browse.html?node=201590011`. Date of last visit: 20th April, 2007.

[17] Martin Erzberger and Marcel Altherr. Every DAD needs a MOM (Message-oriented Middleware), 1999. URL www.softwired.ch/pdf/technology/momdad-final.pdf.

[18] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[19] Globus. The Globus Alliance. URL http://www.globus.org/. Date of last visit: 15th March, 2007.

[20] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press.

[21] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM Syst. J.*, 18(1):111–142, 1979.

[22] JSDL. Job Submission Description Language (JSDL) Specification 1.0, November 2005. URL www.gridforum.org/documents/GFD.56.pdf. Date of last visit: 15th November, 2006.

[23] JVM. Java SE Technologies at a Glance. URL http://java.sun.com/javase/technologies/. Date of last visit: 14th February, 2007.

[24] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner. The Manchester University Atlas Operating System Part I: Internal Organization. *The Computer Journal*, 4(3):222–225, 1961. doi: 10.1093/comjnl/4.3.222. URL http://comjnl.oxfordjournals.org/cgi/content/abstract/4/3/222.

[25] Tom Kilburn, R. Bruce Payne, and David J. Howarth. The Atlas supervisor. pages 49–77, 2000.

[26] Evangelos Kotsovinos. Global public computing. Technical Report UCAM-CL-TR-615, University Of Cambridge, January 2005. URL http://www.xenoservers.net.

[27] libvirt. the virtualization API. URL http://libvirt.org/. Date of last visit: 12th February, 2007.

[28] M2Crypto. MeTooCrypto. URL http://wiki.osafoundation.org/Projects/MeTooCrypto. Date of last visit: 20th April, 2007.

[29] John McCarthy. Reminiscences on the History of Time-Sharing. *IEEE Ann. Hist. Comput.*, 14(1):19–24, 1992. ISSN 1058-6180.

[30] René Meyer and V. Cahill. Taxonomy of distributed event-based programming systems. *The Computer Journal*, 48:602–626, 2005.

[31] MLS. Chapter 3: Implementing Transport and Message Layer Security. URL http://msdn2.microsoft.com/en-us/library/aa480582.aspx. Date of last visit: 22nd April, 2007.

[32] MPI. Message Passing Interface. URL http://www-unix.mcs.anl.gov/mpi/. Date of last visit: 23rd April, 2007.

[33] OGSA-BES. OGSA Basic Execution Service working group. URL https://forge.gridforum.org/sf/projects/ogsa-bes-wg/. Date of last visit: 5th March, 2007.

[34] OpenSSH. Keeping your communiqués secret. URL http://www.openssh.com. Date of last visit: 15th March, 2007.

[35] Alexander Petry. Eventbasierte Simulation von Middleware Plattformen, 2006. Projektarbeit (german).

[36] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. ISSN 0001-0782.

[37] Gerald J. Popek and Charles S. Kline. The PDP-11 virtual machine architecture: A case study. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 97–105, New York, NY, USA, 1975. ACM Press.

[38] POSIX. PASC Collaborative Work Area. URL http://www.pasc.org/plato/. Date of last visit: 10th February, 2007.

[39] POV-Ray. The Persistence of Vision Raytracer. URL http://povray.org/. Date of last visit: 20th April, 2007.

[40] PycURL. PycURL Home Page. URL http://pycurl.sourceforge.net/. Date of last visit: 20th April, 2007.

[41] Python. Python Programming Language – Official Website. URL http://www.python.org/. Date of last visit: 15th March, 2007.

[42] QEMU. Open Source Processor Emulator. URL http://www.qemu.org/. Date of last visit: 14th February, 2007.

[43] RFC2246. The TLS Protocol Version 1.0. URL http://www.faqs.org/rfcs/rfc2246.html. Date of last visit: 15th February, 2007.

[44] RFC2396. Uniform Resource Identifiers (URI): Generic Syntax. URL http://www.faqs.org/rfcs/rfc2396.html. Date of last visit: 15th February, 2007.

[45] RFC2459. Internet X.509 Public Key Infrastructure. URL http://www.faqs.org/rfcs/rfc2459.html. Date of last visit: 15th February, 2007.

[46] RFC4346. The TLS Protocol Version 1.1. URL http://www.faqs.org/rfcs/rfc4346.html. Date of last visit: 15th February, 2007.

[47] R. L. Rivest, A. Shamir, and L. M. Adelman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[48] RMI. Remote Method Invocation. URL http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp. Date of last visit: 22nd April, 2007.

[49] J. Robin and C. Irvine. Analysis of the Intel Pentium's Ability to support a Secure Virtual Machine Monitor, 2000.

[50] Todd Rowland. Church-Turing Thesis – from Wolfram MathWorld, November 2002. URL http://mathworld.wolfram.com/Church-TuringThesis.html. Date of last visit: 20th April, 2007.

[51] RPC. Remote Procedure Calls (RPC). URL http://www.cs.cf.ac.uk/Dave/C/node33.html. Date of last visit: 23rd April, 2007.

[52] Bruce Schneier. *Applied Cryptography, Second Edition*. John Wiley & Sons, 1996.

[53] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972. ISSN 0001-0782.

[54] Amit Singh. An introduction to virtualization. URL http://www.kernelthread.com/publications/virtualization/. Date of last visit: 20th April, 2007.

[55] Stomp. Stomp - Home. URL http://stomp.codehaus.org/. Date of last visit: 20th April, 2007.

[56] Christopher Strachey. Time sharing in large, fast computers. In *IFIP Congress*, pages 336–341, 1959.

[57] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[58] Twisted. Twisted Matrix Labs. URL http://twistedmatrix.com/trac/. Date of last visit: 10th March, 2007.

[59] UNICORE. Uniform Interface to Computing Resources. URL http://www.unicore.org/. Date of last visit: 15th March, 2007.

[60] VMWare. Virtualization, Virtual Machine & Virutal Server Consolidation. URL http://www.vmware.com/. Date of last visit: 10th February, 2007.

[61] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical report, University of Washington, 02 2001.

[62] Wine. Wine HQ. URL http://www.winehq.com/. Date of last visit: 10th February, 2007.

[63] WSS. OASIS Web Service Security (WSS) TC. URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss. Date of last visit: 22nd April, 2007.

[64] Xen. Xen virtual machine monitor. URL http://www.cl.cam.ac.uk/research/srg/netos/xen/. Date of last visit: 19th February, 2007.

[65] XML. Extensible Markup Language (XML), . URL http://www.w3.org/XML/. Date of last visit: 15th March, 2007.

[66] Canonical XML. Canonical XML, . URL http://www.w3.org/TR/xml-c14n. Date of last visit: 15th March, 2007.

Erklärung an Eides Statt

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit mit dem Thema „Design and Implementation of a Xen-Based Execution Environment" selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum)	(Name, Unterschrift)