D2.4.1
TClouds Prototype Architecture, Quality Assurance Guidelines, Test Methodology and Draft API

Project number:	257243
Project acronym:	TClouds
Project title:	Trustworthy Clouds - Privacy and Resilience for Internet-scale Critical Infrastructure
Start date of the project:	1st October, 2010
Duration:	36 months
Programme:	FP7 IP

Deliverable type:	Report
Deliverable reference number:	ICT-257243 / D2.4.1 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.4
Due date:	September 2011 – M12
Actual submission date:	3rd October, 2011

Responsible organisation:	POL
Editor:	Emanuele Cesena
Dissemination level:	Public
Revision:	1.0

Abstract:	This report describes quality assurance guidelines, common use cases, initial architecture, preliminary API and test methodology for the TClouds integrated proof of concept prototype. Moreover, it provides a list of subsystems that shall be developed in next years to demonstrate the results of our research.
Keywords:	Cloud computing, architecture, API, prototype, testing

Editor

Emanuele Cesena (POL)

Contributors

Sören Bleikertz, Christian Cachin, Thomas Groß, Michael Osborne (IBM)

Mina Deng (PHI)

Michael Gröne, Norbert Schirmer (SRX)

Alysson Bessani, Miguel Correia, Marcelo Pasin (FFCUL)

Imad M. Abadi (OXFD)

Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

Johannes Behl, Klaus Stengel (FAU)

Ilaria Baroni, Marco Nalin (HSR)

Paulo Jorge Santos (EFA)

Sven Bugiel, Stefan Nürnberger (TUDA)

Disclaimer

This work was partially supported by the European Commission through the FP7-ICT program under project TClouds, number 257243.

Executive Summary

TClouds, and specifically WP2.4, aims to design a resilient cloud-of-clouds infrastructure, that will be demonstrated by building an integrated proof of concept prototype of a trustworthy cloud environment.

This deliverable reports the work done in the first year within WP2.4 and is organized in two main parts.

The first part describes our methodology and outlines the main results in defining the TClouds architecture and building the TClouds proof-of-concept prototype. In more details, we define actors and common use cases for a cloud infrastructure; describe our efforts to evaluate and decide on an open source cloud computing framework as starting point for our technical development; introduce the initial TClouds architecture; draft the preliminary API of the TClouds platform; define a test methodology to be applied to the TClouds design and development.

The second part contains the set of subsystems that will be developed by partners as part of the integrated proof-of-concept prototype. For each subsystem an overview, including its security goals, selected use cases that define its functional requirements, preliminary high-level architecture and draft API are provided.

The main outcomes of this work are a consistent design of 15 subsystems that will be developed by partners and an initial TClouds platform v0, i.e. an unmodified instance of OpenStack [opeb] – the selected open source cloud computing framework – on top of which a prototype application from Activity 3 is currently running.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

TClouds, and specifically WP2.4, aims to design a resilient cloud-of-clouds infrastructure, that will be demonstrated by building an integrated proof of concept prototype of a trustworthy cloud environment.

The main objectives of WP2.4 can be summarized in designing an architecture for a resilient cloud-of-clouds and building an integrated prototype platform, as a result of the combination of subsystems developed by partners.

More concretely, in the first year of the project, these long term objectives can be better specified as:

- Building a suitable software engineering methodology and devising quality criteria to address the future development.
- Designing the initial high-level TClouds architecture, in conjunction with other WPs, and specifically the high-level TClouds *prototype* architecture.
- Selecting an open source framework for cloud computing as a common platform to be extended by the subsystems.
- Defining a set of subsystems, originating from the research done in WP2.1, WP2.2 and WP2.3, that will be part of the integrated prototype.
- Providing a consistent, high-level design of each subsystem.

In the next section we outline the work done in Y1 within WP2.4 to achieve these objectives. The main outcomes of this work, extensively described in this deliverable, are a consistent design of 15 subsystems that will be developed by partners and an initial TClouds platform v0, i.e. an unmodified instance of OpenStack [opeb] – the selected open source cloud computing framework – on top of which a prototype application from Activity 3 is currently running.

1.1 Outline of the Work Done in Y1

The work done in Y1 within WP2.4 has been organized in phases ended at M2, at M4 (technical meeting in Lisbon), at M8 (technical meeting in Turin) and at M12. In each phase, one or more activities have been carried on, where usually the majority of the partners was involved. Each activity ended with a written report (or activity paper) to consolidate the results, that are collected in this deliverable.

By M2, we drafted the software engineering methodology and we performed a first-round analysis of four open source frameworks for cloud computing, to select a common platform for building the TClouds prototype (Appendix A).

By M4, we consolidated the methodology (Report R2.4.1.1 and Chapter 2), performed an extended analysis of the two most promising frameworks out of the four ones, and selected OpenStack as reference platform (Chapter 4). In addition, a first version of the TClouds architecture has been drafted (Report R2.2.1.1).

By M8 we performed two main activities: the definition of the subsystems and related use cases (Report R2.4.1.2 and Chapter 3) and a preliminary architecture including sequence diagrams (Report R2.4.2.1 and Chapter 4).

Finally, by M12 we finalized the TClouds architecture (Chapter 4), we designed a preliminary API (Report R2.4.2.2 and Chapter 5) and we defined a test methodology (Report R2.4.5.1 and Chapter 6).

All the work on the use cases selection, design of the high-level architecture, draft API and test methodology has been done by each partner on his subsystems, following the common methodology shared along the project. This iterative process led to the description of the subsystems that forms Part II of this deliverable.

1.2 Structure of This Report

This deliverable is organized in three parts: Part I describes our methodology and outlines the main results achieved in Y1, in designing the TClouds architecture and building the TClouds proof-of-concept prototype; Part II collects the list of the subsystems that each partner will deliver as part of the integrated prototype; Part III contains the appendixes.

In more details, Part I is organized as follows. Chapter 2 defines a software engineering process that includes the systematic development, evaluation, and maintenance of cloud components, and discusses the quality criteria that this process has to fulfill. This serves as an introduction to the following chapters that detail the main phases of the development process.

Chapter 3 defines actors and common use cases for a cloud infrastructure. These are the foundation for defining functional requirements of the TClouds prototype, and are extended in several directions by the subsystems detailed in Part II.

Chapter 4 describes our approach towards the high-level architecture for the TClouds platform. It consists of two parts. First, in Section 4.1 we describe our efforts to evaluate and decide on an open source cloud computing framework as a starting point for our technical development. Second, in Section 4.2 we introduce the initial TClouds architecture (see also D2.2.1, Chapter 4) and the subsystems that partners plan to contribute for the TClouds prototype.

Chapter 5 introduces the preliminary API of the TClouds platform. We provide an overview of the OpenStack API, Trusted Infrastructures API and Cloud-of-Clouds API as starting points for our technical development. Moreover, we classify the APIs of the subsystems that constitute the TClouds prototype according to different relevant parameters (type, functionality, client, deployment, ...).

Chapter 6 introduces basic concepts of software testing and explains how they will be applied to the TClouds design and development, in order to meet the functional requirements and to match the desired quality level. We define four layers of testing, namely component, internal API, TClouds API, and application/user interface and we propose a preliminary test plan for each subsystem that constitute the TClouds prototype (for the application/user interface layer, we also provide an overview of the test plan from Activity 3).

In Part II, each chapter describes a subsystem that will be developed as part of the integrated proof-of-concept prototype and it is organized with: an overview of the subsystem, including its security goals; selected use cases that define its functional requirements; preliminary high-level, and possibly low-level, architecture; draft API.

Finally, Part III is organized as follows. Appendix A contains the template and results of the first-round analysis performed to select an open source cloud computing framework as common platform for building the TClouds prototype.

Appendix B contains a list of open source cloud frameworks, testing tools and frameworks, and public cloud services that are referenced within this deliverable.

For a glossary of the technical terms used within this report, we refer to D2.1.1.

Part I

TClouds Prototype Architecture, Quality Assurance Guidelines, Test Methodology and Draft API

Chapter 2

Quality Assurance Guidelines

Chapter Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

In this chapter we define a software engineering process that includes the systematic development, evaluation, and maintenance of cloud components, and we discuss the quality criteria that this process has to fulfill. For a more ample discussion on software and system engineering we refer to Endres and Rombach's book [ER03].

2.1 Quality Criteria

The quality of the TClouds platform cannot be expressed with a single parameter or a simple statement, but is the combination of several factors, which include dependability and security properties, as well as legal and economical aspects. For the last ones, we refer to the work done in Activity 1, in more details to D1.2.2 for legal aspects and to D1.3.1 for business requirements.

In the context of this report, we define the *quality* of the TClouds platform as the degree to which it meets stakeholders' requirements. Stakeholders include of course cloud (TClouds) users, but also developers, as the TClouds platform will be built upon an existing open source framework, thus part of its contribution may be released to the community in the future.

In the following we introduce the main quality criteria that will be used to evaluate the TClouds platform.

2.1.1 Quality for TClouds Users

The quality of the TClouds platform from a user's point of view can be expressed as the fulfillment of several properties:

Availability: High degree of access.

Reliability: Low failure rate.

Efficiency: Economic resource consumption.

Usability: Well adapted to skills and preferences of user.

Robustness: Safe reaction to user errors and hardware failures.

Security: Low damage in case of negligent/malicious use.

Since *security* is the distinguish feature of TClouds with respect to commodity clouds, we further refine it by defining the following properties:

Fault tolerance: Operational continuity (or graceful degradation) in case of failure of some components.

Data integrity: Guarantee that data has not been altered (modified, deleted, duplicated...) by unauthorized users.

Data confidentiality: Guarantee that data cannot be accessed by unauthorized users.

Data authenticity: Guarantee that data has been created by an identified user.

Privacy protection: Guarantee that sensitive data is not exposed to undesired parties.

2.1.2 Quality for TClouds Developers

As already mentioned, the TClouds prototype will build upon an existing open source framework for cloud computing, and one of the possible outcomes of the project is to enhance the features of the base framework with security-enhanced and/or privacy-enabled components.

Thus, next to the user's criteria, also developer-oriented properties can be considered:

Installability: Easy and fast setup.

Testability: Good documentation and structure.

Maintainability: High readability and modifiability.

Portability: Low dependency on technical environment.

Localizability: Adaptable to national and regional requirements.

Reusability: High modularity, completeness and low coherence. This is especially important to allow integrating different components (and models) in a single prototype (see Sec. 2.2 for more details).

2.2 Software Engineering Process

The TClouds prototype should go through a lifecycle of its own, which proceeds in parallel to the system lifecycle as shown in Figure 2.1.

Each partner will provide technical contributions that we shall refer to as *subsystems* (of the TClouds prototype). We will use the term *component* to refer to parts of each subsystem or of the cloud framework more in general.

We now describe the software engineering process that guided and will guide the development of the subsystems. Firstly, a subset of the use cases originating from the application scenarios has to be selected and analyzed, and this will represent the set of functionality demonstrated by the prototype. Secondly, an architecture and the related sequence diagrams and API will be defined, based on the selected use cases. Next, the prototype will be developed, building upon a selected open source framework for cloud computing. The subsystems that will be part of the prototype will be developed by each partner independently as part of the research activity,

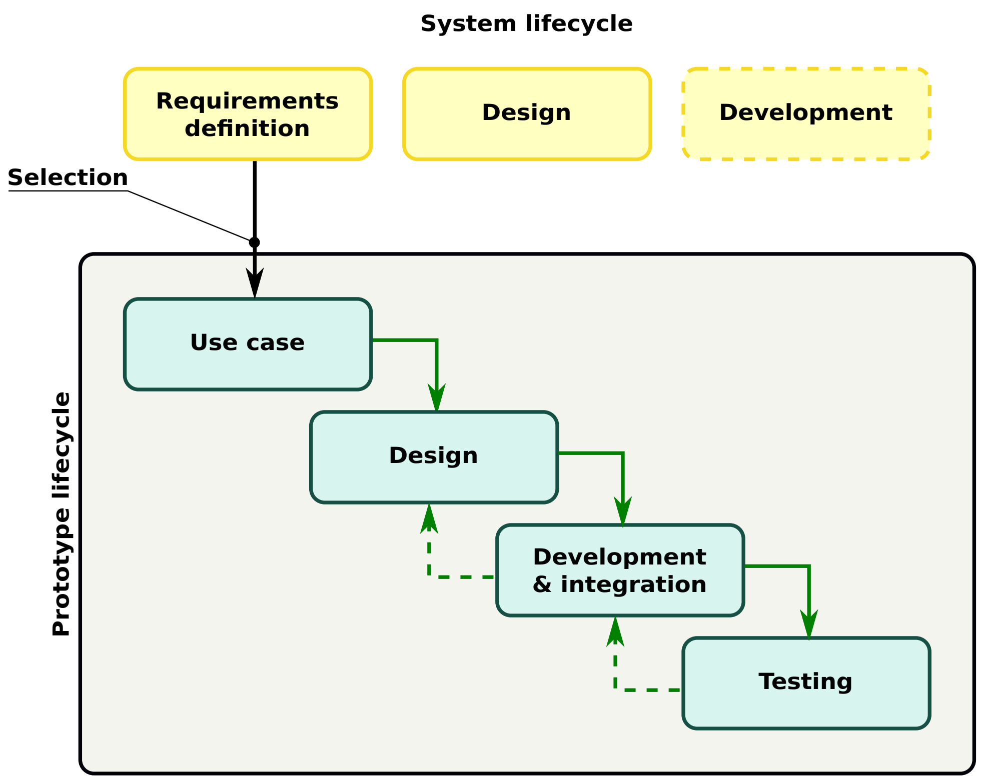


Figure 2.1: TClouds prototype in the system lifecycle

and will be integrated in the TClouds prototype. Finally, the prototype will be evaluated, i.e. tested, and this requires the definition of a suitable methodology.

The prototype will follow an iterative project cycle. As an overall development strategy, we agreed on the following timeline. By the end of Y1, we will have a TClouds platform v0, i.e. an unmodified instance of the selected open source framework. A few subsystems will be shown to demonstrate the effectiveness of the research developed in Y1. By Y2, we expect to incorporate a few application-specific functionalities into the TClouds platform, as a result of the research activity. This practically means that some subsystems will be integrated in the selected framework, and this is referred to as first mock-up integration, or TClouds platform v1. Finally, by the end of the project the final TClouds prototype (TClouds platform v2) will be implemented, supporting all the functionalities required by the selected use cases.

2.2.1 Use Cases Selection and Analysis

Functional requirements will be specified and detailed by means of use cases. Because of the complexity of the cloud environment, only a subset of the use cases will be prototyped.

In more detail, we will first define common use cases, derived from application scenarios and commodity clouds. Next, a subset of these use cases will be selected and detailed, providing new and/or enhanced functionality. The selection has to meet the following conditions:

- Consistency: the selected use cases must not depend on other, undefined, use cases.
- Coherence: the selected use cases, when read back as a portion of the application scenarios, still have to represent a meaningful application.

- Completeness: the selected use cases actually highlight the distinguishing features of TClouds.

Finally, use cases will be analyzed to guarantee that the properties mentioned above are satisfied. The resulting set of use cases will form the basis for the specification of the TClouds prototype architecture.

Functional requirements have to be complemented by non-functional ones, that may address such quality criteria as reliability, efficiency and usability. Other criteria like portability, testability and maintainability can be of particular interest when the focus is the cloud framework as a software product that can, for instance, be released to the open source community. Non-functional requirements typically conflict with each other, and with the functional requirements. When this happens, a trade-off possibility should be specified.

A specific class of non-functional requirements concern the safety and the security of a system. Security requirements (and security assumptions) are of main importance for the TClouds project as trustworthiness is the distinguishing feature of the TClouds framework with respect to other similar products. A clear definition of trustworthiness and detailed security requirements will be specified. It is crucial to bound security requirements to use cases and to address the potential threats and risks.

2.2.2 Architecture Definition and Related API

The TClouds prototype architecture will be designed, based on the selected use cases, and models for the system and each subsystem will be specified. Use cases will then be mapped onto the architecture to derive the workflow among components, specified, e.g., with sequence diagrams.

The architecture will be prototyped building upon an existing open source framework for cloud computing. A specific activity has been devoted to survey currently available open source frameworks and select a suitable candidate. The activity is organized in two phases, described below.

In the first phase, ended on M2, we surveyed the following 4 framework: Nimbus [nim], Eucalyptus [euc], OpenNebula [opea], OpenStack [opeb]. The template used for this first analysis is reported in Appendix A. As a result of this phase, we selected 2 candidates for further analysis: OpenNebula and OpenStack.

In the second phase of the analysis, we installed and tested extensively the two platform candidates. As a result of this phase, we prepared a tutorial on how to deploy a cloud-oriented application onto a cloud infrastructure, and two technical talks describing the architecture of OpenNebula and OpenStack, as well as the supported APIs. The tutorial and the talks have been presented at the TClouds technical meeting at M4 in Lisbon, where OpenStack was also selected as reference platform to build TClouds. Further details on these activities are given in Chapter 4.

The TClouds architecture will be completed by the definition of 2 APIs: the *TClouds API* and the *internal API*. The TClouds API will extend a selected public API, notably by introducing security enhanced concepts and features. The internal API is used to provide an explicit interface for relevant functionalities, that can be either related to critical components or useful when more components exist that implements a particular functionality. In all cases, the internal API is useful for testing purposes, to decouple the test definition and the component implementation.

2.2.3 Test Methodology

A test methodology will be set to assist the development of the TClouds prototype. Test cases will be defined for all relevant subsystems and components based on the selected use cases. Then the test cases will be applied first to the initial mock-up integration, then to the final platform prototype.

We define four layers of tests: component, internal API, TClouds API and application tests.

Component tests are optional. Each partner may release its components together with a fully automatic testsuite that, from the perspective of the TClouds project, plays the role of unit testing. Components testsuites can be integrated in a single testing framework, for instance Hudson [hud] or Jenkins [jen] (also used by the OpenNebula project).

Internal and TClouds API tests are similar, while they apply to two distinct interfaces. Internal API tests are black-box tests for TClouds subsystems, while TClouds API tests correspond to integration tests. The API coverage should be close to 100% in terms of number of functions tested, for at least one fixed choice of function parameters and a default platform configuration. These tests will be automatic, e.g. with REST interface and Selenium [sel], with the possible exception of the platform configuration that may require manual intervention to be changed.

Application tests correspond to the tests of the two prototype applications, that we shall develop in A3 to demonstrate the functionality of TClouds. These include the interaction with the TClouds API, that should be a subset of the TClouds API tests, and the graphical user interface. Both tests should be as automatic as possible, e.g. with Selenium or SIKULI [sik].

Chapter 3

Use Case Selection

Chapter Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

In this chapter we define actors and common use cases for a cloud infrastructure. These are the foundation for defining functional requirements of the TClouds prototype, and will be extended in several directions by the subsystems detailed in Part II.

3.1 Actors

Actors and components defined in this report map to OpenStack's architecture. When possible or known, the corresponding Amazon Web Services (AWS) names are also given.

- *User*: The end-user of the cloud services. User can be further specialized in *Project Manager* or *Developer*. The former owns a project, or TVD[1], and has administrative privileges on the project; the latter has access to some resources within a project and his privileges are defined by the project manager.
- *Cloud*: A cloud infrastructure. It is useful to look at the cloud as a whole, mainly for scenarios related to the Cloud of Clouds.
- *Cloud Node*: A physical node of the cloud infrastructure. A node can be further specialized in *Computing Node*, *Storage Node*, *Network Node*, *Object Store Node*, *Image Node* and *Management Node*.
- *Cloud Component*: A component of the cloud infrastructure, intended as a service that can be either provided to the User or internally used by the Cloud itself, i.e. by other components. It is useful to refer to a Cloud Component when multiple Cloud Nodes cooperate to provide a service. Here we introduce the following specializations: *Storage Component*, *Object Store Component*, *Image Component* and *Management Component*. Other Cloud Components will be introduced in next sections.
- *Management Component*: The collection of services used to administrate the cloud infrastructure. It includes: *Scheduler*, *Queue*, *Network Component*, *API Component* and *Management Console* (also referred to as Control Panel). This component is crucial in the TClouds design and will be enhanced in WP 2.3.

[1]Projects are isolated resource containers forming the principal organizational structure within Nova. In TClouds, an OpenStack project is usually referred to as Trusted Virtual Domain (TVD).

The Management Component, in general, provides functionality both to the Project Manager and to the Cloud Admin. Whenever necessary, we distinguish between *Management* and *Administration* to denote respectively functionality for the Project Manager or the Cloud Admin.

- *Cloud Admin*: The administrator(s) of the cloud infrastructure. More generally, Cloud Admin MAY also refer to the Management Component when it is not important to distinguish between automatic procedures or operations that require human intervention.

3.2 Common Use Cases

We introduce the following terminology:

- *VM instance*: is a ready-to-run, or running, VM. It can be persistent or not.
- *VM image*: is a "bundled environment" that includes all the necessary bits to set up and boot a VM instance. This is usually stored in a object storage for availability reasons and copied into a disk volume when a VM instance is created. In AWS terminology, this is called Amazon Machine Image (AMI).
- *Volume*: is a block level storage for use with a VM instance. Each instance has at least one volume, which is initially copied from the VM image but, more in general, the instance can have more volumes attached. A volume can only be attached to a single VM instance. In AWS terminology, this is called Elastic Block Store (EBS) volume.

For some use cases in the "Alternative Flow (Similar use cases)" box there is a list of complementary use cases (e.g. stop vs. start, etc.), whose description is omitted for simplicity.

3.2.1 Computing

The first use case (*Create Instance*) is given in detail. In the followings, for simplicity, the Management Component is not included in the actors and the flow is omitted.

USE CASE UNIQUE ID	**/UC 10/ (Create Instance)**
DESCRIPTION	User creates a new VM instance.
ACTORS	User, Management Component, Image Component, Computing Node. Storage Component and Object Store Component may also participate.
PRECONDITIONS	None.
POSTCONDITIONS	A new VM instance owned by User is created on Computing Node.
NORMAL FLOW (NON-PERSISTENT)	1. User requests the creation of a new VM instance to Management Component. 2. Management Component schedules a Computing Node to host the VM instance and relies on the Image Component to create the instance from the VM image (template). 3. Image Component copies the VM image (optionally from the Object Store Component) into the Computing Node.
ALTERNATIVE FLOW (PERSISTENT)	3'. Image Component copies the VM image (optionally from the Object Store Component) into the Storage Component.

USE CASE UNIQUE ID	**/UC 20/ (Start Instance)**
DESCRIPTION	User starts a VM instance.
ACTORS	User and Computing Node. Storage Component and Network Component may also participate.
PRECONDITIONS	User created the VM instance (cf. /UC 10/).
POSTCONDITIONS	The VM instance is running on Computing Node.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Stop Instance 2. Reboot Instance 3. Terminate Instance (or Destroy Instance)

3.2.2 Image and Volume Storage

USE CASE UNIQUE ID	**/UC 30/ (Create Image)**
DESCRIPTION	User creates a new VM image.
ACTORS	User and Image Component. Object Store Component may also participate.
PRECONDITIONS	None.
POSTCONDITIONS	A new VM image owned by User is created on Storage Node.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Delete Image 2. Retrieve Image 3. Update Image

USE CASE UNIQUE ID	**/UC 40/ (Create Volume)**
DESCRIPTION	User creates a new volume.
ACTORS	User and Storage Component.
PRECONDITIONS	None.
POSTCONDITIONS	A new volume owned by User is created on Storage Component.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Delete Volume

USE CASE UNIQUE ID	**/UC 50/ (Attach Volume)**
DESCRIPTION	User attaches a volume to a VM instance.
ACTORS	User, Computing Node and Storage Component.
PRECONDITIONS	User created the VM instance (cf. /UC 10/) and the volume (cf. /UC 40/).
POSTCONDITIONS	The VM instance has a new volume attached.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Detach Volume

USE CASE UNIQUE ID	**/UC 60/ (Access Volume)**
DESCRIPTION	Attached volume is used by an instance.
ACTORS	Computing Node
PRECONDITIONS	Volume is attached to the VM instance (cf. /UC 50/).
POSTCONDITIONS	The VM instance stores data on or read data from the attached volume.

3.2.3 Object Storage

USE CASE UNIQUE ID	**/UC 70/ (Create Name Space)**
DESCRIPTION	User creates a new name space.
ACTORS	User and Object Store Component.
PRECONDITIONS	None.
POSTCONDITIONS	A new name space owned by User is created on Object Store Component.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Delete Name Space

USE CASE UNIQUE ID	**/UC 80/ (Write Object)**
DESCRIPTION	User writes an object.
ACTORS	User and Object Store Component.
PRECONDITIONS	User owns a name space (created via /UC 70/).
POSTCONDITIONS	A new object within the name space is created on Object Store Component.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Read Object 2. Delete Object

3.2.4 Administration

USE CASE UNIQUE ID	**/UC 90/ (Login)**
DESCRIPTION	Cloud Admin logs in to a Cloud Node, e.g. via SSH.
ACTORS	Cloud Admin and Cloud Node.
PRECONDITIONS	None.
POSTCONDITIONS	Cloud Admin has root access to Cloud Node.

USE CASE UNIQUE ID	**/UC 100/ (Migrate Instance)**
DESCRIPTION	Cloud Admin migrates a VM instance from Computing Node X to Computing Node Y.
ACTORS	Cloud Admin, Computing Node X and Computing Node Y. Storage Component and Network Component may also participate.
PRECONDITIONS	The VM instance is running on Computing Node X.
POSTCONDITIONS	The VM instance is running on Computing Node Y.

Chapter 4

Initial Prototype High-Level Architecture

Chapter Authors:
Sören Bleikertz (IBM)
Michael Gröne, Norbert Schirmer (SRX)
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

This chapter describes our approach towards the high-level architecture for the TClouds platform. It consists of two parts. First, in Section 4.1 we describe our efforts to evaluate and decide on an Open Source Cloud Computing framework as starting point for our technical development. Second, in Section 4.2 we introduce the initial TClouds architecture (see also D2.2.1, Chapter 4) and the technical artifacts, i.e. subsystems, that partners plan to contribute for the TClouds prototype. Details on each subsystem are then given in Part II.

4.1 Framework Evaluation and Selection

The TClouds architecture will be prototyped building upon an existing open source framework for cloud computing. A specific activity is devoted to survey currently available open source frameworks and select a suitable candidate. The activity is organized in two phases, described below.

In the first phase, ended on M2, we surveyed the following 4 framework: Nimbus [nim], Eucalyptus [euc], OpenNebula [opea], OpenStack [opeb].

Table 4.1 shows minimal criteria used for the decision. The outcome of the this first analysis is reported in Appendix A.

	Storage Support	Cloud of Cloud Support	Driving Community
Eucalyptus	x		academic
OpenNebula		x	academic
OpenStack	x		industry
Nimbus	x		academic

Figure 4.1: Framework evaluation: excerpt of decision matrix

As a result of this phase, we selected 2 candidates for further analysis: OpenNebula and

OpenStack. The former represents an academic project which already supports a cloud of cloud scenario, the latter is an industry driven approach which already offers an object store API.

In the second phase of the analysis, we installed and tested extensively the two platform candidates. As a result of this phase, we prepared a tutorial on how to deploy a cloud-oriented application onto a cloud infrastructure, and two technical talks describing the architecture of OpenNebula and OpenStack, as well as the supported APIs. The tutorial and the talks have been presented at the TClouds technical meeting at M4 in Lisbon, where OpenStack was also selected as reference platform to build TClouds.

In the following we describe in more detail the evaluation process performed to select the platform candidate.

The criteria used for the evaluation are:

- **Deployment:** this evaluates the simplicity of the platform installation in common Linux distributions using the provided documentation.
- **Modularity:** this evaluates the internal software structure (monolithic or modular), in order to verify if the platform can be extended with TClouds developed services and existent components can be replaced without breaking any functionality.
- **API Interface:** this evaluates the API available for developers to create their applications.
- **Community Activity:** this evaluates if the software is currently maintained and how useful is the support provided by developers through common channels (web site, mailing lists).

4.1.1 OpenNebula

4.1.1.1 Architecture and Installation

We evaluated OpenNebula v2.0.1, whose architecture is depicted in Figure 4.2.

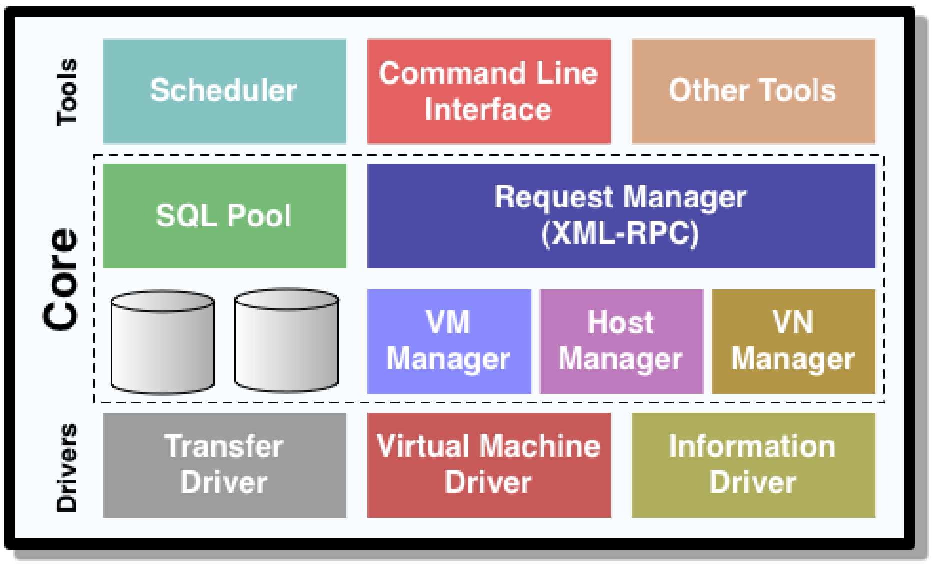


Figure 4.2: OpenNebula architecture overview (source: OpenNebula web site [opea])

At the highest level there tools distributed with OpenNebula, such as the *Command Line Interface* (CLI), the scheduler, or the Cloud RESTful interfaces (currently Amazon EC2 and OCCI APIs are supported). All these tools communicate with the OpenNebula core through the XML-RPC interface or the new OpenNebula Cloud API (OCA).

At lower level, the OpenNebula core is split into several functional components which are responsible to implement methods defined for the above mentioned objects. The *Request Manager* handles client requests, the *Virtual Machine Manager* manages and monitors virtual machines, the *Transfer Manager* manages virtual machine images, the *Virtual Network Manager* manages virtual networks, the *Host Manager* manages and monitors physical resources and, finally, the *Database* persistently stores OpenNebula data structures. The interaction between these components and a particular hypervisor or a file transfer mechanism is possible through a set of modules called *Drivers*.

The OpenNebula core exposes a rich object-oriented interface, the OCA, to manage all cloud resources. The objects defined are: *image*, *virtual machine*, *virtual network*, *user*, *host* and *cluster*, each with a set of specific methods. A new virtual machine, for instance, can be started by calling the methods *allocate* and *deploy* on the instantiated object, or can be migrated to another cluster node by invoking the method *migrate*.

The installation process was performed using the documentation located at the OpenNebula's website [opea] upon several Linux distributions, either from packages, when available, or compiling from the sources in the other cases. While the installation process has been straightforward, the requirements in terms of other system components such as *libvirt* and *KVM* were critical. In particular, the Linux distributions Debian Lenny, CentOS 5.5 and Ubuntu 10.10 contain a version of these components that is not compatible with OpenNebula. In the end, we chose RedHat Enterprise 6 beta2 Linux as distribution to perform extended tests.

During the installation we found a critical issue using the EC2 interface, as all commands sent were rejected with an authentication failure message [Smi11]. The problem was caused by an incompatibility between the client-side library (shipped with the RHEL6 distribution) and the OpenNebula EC2 server, as they used different digest algorithms for authenticating messages.

4.1.1.2 Extended Evaluation

The test cloud used in this evaluation consists of 3 nodes: one node acting as *frontend* and containing the OpenNebula software plus the SQL server (MySQL); two other nodes, called generically *nodes*, executing virtual machines, that rely on *libvirt* and a hypervisor (only the *KVM* hypervisor has been tested).

One goal of the test installation is to evaluate the *Transfer Manager* component, in particular the driver that implements the *Logical Volume Manager* (LVM) support for the virtual machines storage. The current version of OpenNebula supports only a basic scenario were virtual machine images are stored into LVM logical volumes of each cluster node.

OpenNebula supports two options for the storage management: a *shared storage* and a *non-shared storage*. In the former solution virtual machine image templates and instances are stored in a centralized storage repository which can be accessed concurrently by the front end and other cluster nodes. In the latter, an image instance is transferred from the front end, which contains all templates, to the target node before the virtual machine is started.

These two options present advantages and disadvantages in term of performance and fault tolerance. The former solution offers good performance during the virtual machine deployment because there is no data transfer between nodes but only a little I/O overhead during its life cycle. The latter solution introduces a noticeable delay in the virtual machine deployment, because its image is transferred across the network to the target node, but no additional I/O overhead occurs during the life cycle. Regarding the fault tolerance, the shared storage has the issue that there is one point of failure, which means that all virtual machine image templates and instances become unavailable if the node providing this service goes down. In the other case, virtual machines can be still used in the event of a node failure. Further, if the *front end* is not available new virtual machines cannot be created but those already deployed can still provide their services.

In order to verify the flexibility of the OpenNebula architecture, a new *Transfer Manager* driver has been developed to support LVM on top of a shared storage repository. We chose to use the snapshot feature of LVM to decrease the creation time of a virtual machine image instance from the template and to optimize the storage usage by recording only modified data. We note that there exists a version of LVM specifically designed to run in a cluster environment. The system service called `clvmd` [Red10], in conjunction with a distributed locking mechanism, offers the necessary support to concurrent writes on LVM metadata. However, when used with the clustering extension, LVM currently does not support the snapshot feature.

The access model implemented by OpenNebula permits to make some assumptions about the usage of LVM in a cluster environment that can make the distributed locking mechanism not necessary. First, LVM is configured only by one actor (the front end node), which means that metadata can be safely modified because there is only one writer. Second, image templates (origin of the snapshots) are never modified when virtual machines are executed on cluster nodes. This avoids metadata corruption because the origin of the snapshot and the snapshot itself cannot be modified at the same time.

The storage repository used in the test cloud is shared through iSCSI, by configuring one cluster node as iSCSI target and letting the others access it with an iSCSI initiator client [Tim09]. In order to add fault tolerance to the storage repository the exported block device is also replicated to another cluster node using the *Distributed Replicated Block Device* (DRBD) [Lin08].

The implemented driver (`tm_sharedlvm`), written in BASH, issues LVM commands to the front end node, in order to create or remove LVM logical volumes, and to other cluster nodes to update the LVM configuration. This new driver was submitted to the mailing list *Opennebula Users*. The introduction of this new *Transfer Manager* driver does not break any existent functionality, and it can be easily disabled or removed without modifying the original OpenNebula code.

Another goal of the evaluation activity has been to inspect the OpenNebula internal API, called *OpenNebula Cloud API* (OCA), and to verify how simple is to develop an application upon it. One important aspect of the API is that it can be used regardless the programming language used, because messages between the application and OpenNebula are encoded using the XML-RPC transport mechanism. Currently there exists two language bindings for the OCA: Java and Ruby.

One limitation discovered in these API was that images could not be uploaded and stored as LVM logical volume, as only the files were currently supported. However we were able to fix this issue by slightly modifying the OpenNebula code, without affecting the API.

As a final comment on the mailing lists activity, in about two months of testing and experimenting with OpenNebula, we submitted about 25 messages to the developers and always received prompt responses (in this period, the mailing list counted more than 500 messages).

4.1.1.3 Tutorial on OpenNebula

The TClouds project organized an OpenNebula Hands-on during its meeting in Lisbon with the main goal of introducing this cloud resource manager, through a cloud user perspective.

In the beginning, the OpenNebula main components, their interactions and interfaces were presented. The participants were then invited to request the creation of virtual machines, as well, using and removing them from the virtual machines pool. For this hands-on was created a cloud infrastructure with one front-end node and seven physical hosts available to allocate the virtual machines.

4.1.2 OpenStack

OpenStack is an open source Infrastructure as a Service (IaaS) cloud computing platform initiated by RackSpace and NASA. The initiative is joined by other major vendors and startups in the field of cloud computing, such as Citrix, Dell, Cloudkick, AMD, and Intel. The code base of the platform is written in Python (about 70k LOC) and is licensed under the Apache open source license. The mantra of this initiative is that it "strives to become the open source standard for building cloud infrastructures everywhere".

4.1.2.1 Community

The OpenStack project started in July 2010 and already consists of an active and diverse community as represented in the OpenStack Design Summit 2010 where 250 people from 12 countries participated. The development cycles of the project are very short and new releases are planned every 3 to 6 month. In February 2010 a stable version was released that is suitable for mid-sized deployments with regard to compute resources and production ready for storage resources. The April 2010 release is planned to be production ready for large scale service providers.

The community is very diverse and consists of contributors from a variety of organization. There seems to be no single organization behind the project, although it was initiated by RackSpace and NASA that are still major drivers in the project. Over 25 companies are supporting the project and they adopted an open integration process for changes to the code base, i.e., improvements to the code base from new contributors are welcomed and accepted.

The project incorporates professional development practices in order to ensure good quality of the software. Among these practices are unit tests, code reviews, code documentation, and continuous integration. Furthermore, the development and planning of the project are transparently conducted on the open source software collaboration platform *launchpad.net*, which is also used for the development of the popular Ubuntu Linux distribution.

4.1.2.2 Nova: Compute Cloud

Nova contains all the management components that are required to build a compute cloud. It is similar to Amazon EC2 and is based on NASA's cloud project.

Installation: In our evaluation we performed a test installation of Nova on a single machine with a local object store for the virtual machine images. Ubuntu Linux 10.04 and 10.10 contains packages for Nova that can easily be installed. We only encountered minor problems during the installation and we were able to start and log into a virtual machine within 30 minutes.

However, our single-machine installation is much simpler than a multi-machine installation that is required in a real production use case and which would require more work to set up.

Architecture Overview: Figure 4.3 gives an overview of the architecture of Nova. Three controllers manage the resources such as compute, network, and storage: compute worker, network controller, and volume worker. API endpoints collect requests from cloud consumers and a scheduler dispatches requests to the appropriate controller or worker. A queue is used for all message-based communication between the services.

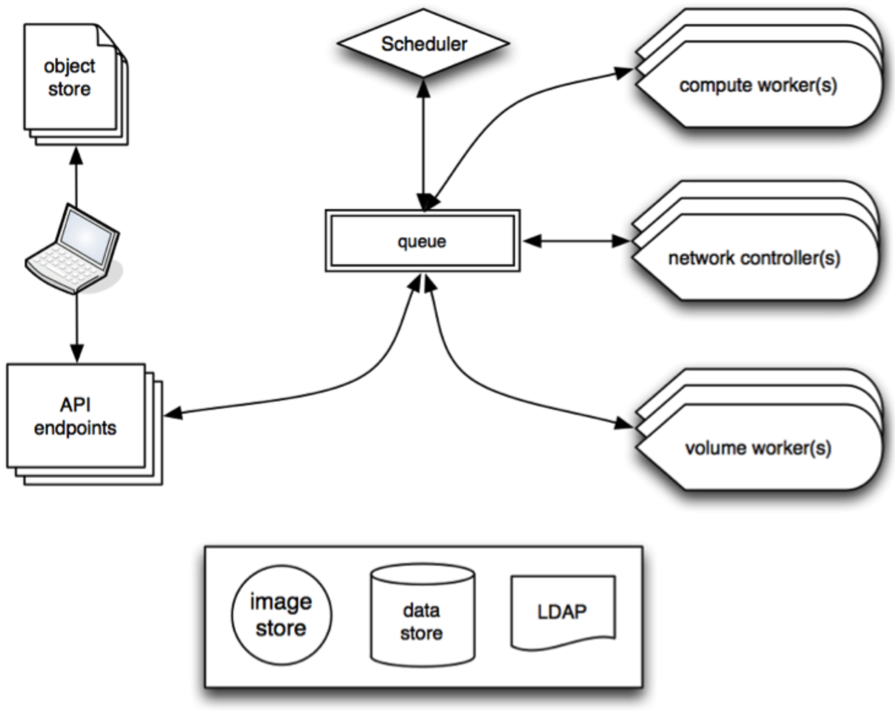


Figure 4.3: OpenStack Nova Architecture (source: OpenStack Nova web site [Oped])

Service Communication: The communication of the services in the Nova architecture is realized using message queues. Running all the services separated from each other and only allow communication using queues reflects the design principles of the Nova architecture: A shared-nothing and messaging-based architecture, the state is held in a distributed data store, and asynchronous calls with call-backs. The message queues are based on the *Advanced Message Queuing Protocol (AMQP)* and Nova uses the *RabbitMQ* implementation of these queues, which is written in Erlang and supports high availability and clustering. Within Nova, message queues are used in a Publish-and-subscribe fashion, where services listen on specific channels, and other services can write to these channels. Furthermore, it is possible to send message to specific hosts, e.g., in order to start a VM on a specific host X.

Cloud API: Nova currently supports two APIs: Amazon EC2 and OpenStack. For EC2, a subset of the API is implemented and the open source management tools, such as euca2ools, can be re-used for managing a Nova compute cloud. Furthermore, the *Open Cloud Computing*

Interface (OCCI) API is planned to be implemented. The web service providing the APIs is also responsible for authentication and authorization of cloud consumers and their requests. The API server dispatches requests via message to the appropriate services in the architecture.

Scheduler: The scheduler is responsible to choose a host to run instances on when the API server dispatches a message that a VM is requested to be started. After selecting a host, the scheduler will forward the request to the selected host, which then can start the requested VM. Currently, multiple scheduler drivers exists: *Chance*, which randomly selects a host; *Simple*, which chooses the host with the least load. A few problems exists with the current scheduler implementation, which will be addressed by the OpenStack developers in the feature. Namely, that only a single scheduler can exist in the Nova architecture, which forms a single point of failure, and that the state is kept in a central data store. For the future, a distributed scheduler is planned, which overcomes these scalability and availability problems.

Compute Worker: The compute worker handles the compute resources on a physical machine. The worker builds disk images for VMs, launches or terminates VMs using a virtualization driver (currently LibVirt and Xen are supported), monitors VM states, attaches or detaches persistent storage volumes to VMs, and provides console outputs from VMs.

Network Controller: The network controller manages the network resources on a host. It basically configures networks and VLANs on a physical machine, but it is not able to configure the network infrastructure such as switches, e.g., in order to setup VLANs on the switch. There exist three different modes for fixed IP addresses: *Flat Mode*, which is a bridged setup that statically allocates IP addresses; *Flat DHCP Mode*, which is also bridged but uses DHCP instead of static IP addresses; *VLAN DHCP Mode*, which uses VLANs in order to provide stronger isolation and provides a VPN gateway to the cloud consumer. Besides fixed IP addresses, there also exists the concept of floating IP addresses that can be dynamically assigned to virtual machines, e.g., in order to assign a static public IP address to a VM.

Volume Worker: The volume worker manages persistent storage volumes that are exported to other hosts. The worker can create and delete LVM-based volumes on a physical host. These volumes can be exported using either iSCSI or ATA over Ethernet (AoE) to hosts, which are running VMs that requested a volume to be attached. Machines hosting volumes for other machines are a single point of failure, and a distributed block storage is needed for resilience. OpenStack developers are looking into adopting Sheepdog, a distributed storage system for QEMU, for this purpose.

Security Goals & Features: OpenStack aims at providing a redundant shared-nothing architecture for scalability and fault tolerance, which is realized except for the central scheduler. Furthermore, OpenStack currently provides a variety of security features. Security groups are a firewall concept introduced by Amazon EC2 that allows to group virtual machines together, e.g., based on functionality, and protect this group with a set of firewall rules. A VLAN-based network setup allows stronger isolation of virtual machines on the network level, and VPN provides a secure access for the cloud consumer to these instances. Authentication is realized using access and secret keys for signing web service calls and X509 certificates for bundling images. Authentication can also be integrated with LDAP. Role-based Access control is implemented for web service calls. Quotas limit the resource usage of cloud consumers and audit logs are planned to be implemented in the future. Images can be uploaded in an encrypted and authenticated form, in order to prevent manipulation and eavesdropping while transferring the image.

4.1.2.3 Swift: Storage Cloud

Swift is a distributed data blob storage system similar to Amazon S3. This component of OpenStack originates from RackSpace's production cloud storage system and is also used in production. With regard to the CAP theorem, Swift only provides eventual consistency.

Installation: Similar to Nova, we performed a single-machine installation using Ubuntu Linux and we used the Ubuntu packages from the developer repository of Swift. Since these packages do not come with default configuration files, setting up Swift took considerably more time than setting up Nova, i.e., about 2 hours.

Architecture Overview: The architecture of Swift is illustrated in Figure 4.4. We can identify three components related to storage: *Account*, *Container* and *Object* servers with their associated rings. The servers store the actual content and the rings are acting as an address book in order to locate the server hosting specific content. For the servers we observe a hierarchy in the stored content, namely that accounts indexes containers and they contain the actual data objects. A *Proxy* mediates all requests and responses between the servers and the users, and furthermore it uses an authentication and authorization service to validate the user's API calls. *Memcache* is used to cache certain responses within the system.

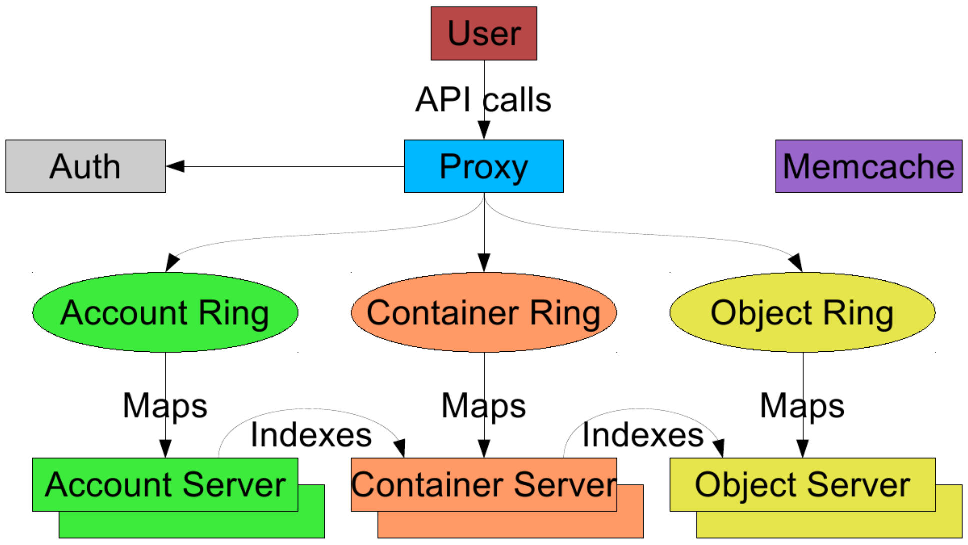


Figure 4.4: OpenStack Swift Architecture

Proxy & Auth: The *Proxy* exposes a ReSTful API to the client and dispatches all incoming requests to the corresponding servers. The API is not S3-compatible, although a S3 API middleware is developed that provides such compatibility. Objects, which are requested by the client, are streamed through the proxy from the object server to the client. For authorization and authentication, there currently exists two services: *DevAuth* and *Swauth*. The former allows external auth services to be plugged in, and the later is a scalable auth service based on Swift itself.

Rings: A *Ring* is a mapping of an entity name to its physical location. In the Swift architecture we have separate rings for the different content types, i.e., account, container, and object. Any action on an entity requires to query the ring in order to locate it. The mapping takes into account zones, devices, partitions, and replicas. Each replicate resides in a different zone, which could be a server, rack, or data center. Partitions are replicated and balanced across the cluster. Devices are used for handoff in failure scenarios and partitions are assigned to devices. A ring is a statically constructed data structure (using the ring-builder) and distributed to the servers. In case the configuration of the Swift system changes, rings are recreated and distributed.

Account & Container Server: The functionality of the account and container servers are very similar. Both store an index of the containers or objects respectively in sqlite database files. Replication of these files to other physical locations is performed by first performing a hash comparison between the source and destination files. If the hashes differ, the records added since a last synchronization point are shared.

Object Server: The object server stores the actual simple data blobs. Objects are stored as binary files with metadata in the file system's extended attributes (xattr). The path of the object file is a combination of the object name's hash and a timestamp. In case an object gets deleted, a special "tombstone" file is placed instead of the file, which is also replicated to the other servers, therefore ensuring that other replica do not serve the deleted file. In case a new version of an object is stored, the older version will be deleted. Large objects are supported with basically infinite size (although depending on the storage cloud capacity) by using client-side chunking that splits the large object into smaller (up to 5 GB) chunks. Replication is done by using *rsync* and pushing data to replica servers. For efficiency reasons, a partial rsync based on hash invalidation is performed.

Updaters & Auditors: There exist two processes which are performed periodically: *Updater* and *Auditor*. The updater updates the index of account or container servers, in case a new container or object is added respectively. Such an update might fail and the update task is queued for later processing, which leads to an eventual consistency window. For example, during this window, a newly added object can be retrieved, but will not be listed in the container. The auditor is responsible for checking the integrity of objects, containers, and accounts. The integrity check is based on a hash comparison for objects, and trying to obtain database information from the sqlite files in case of container and account servers. In case of corruption, the corrupted entity is quarantined and replaced with one from a replica.

Security Features: The main security goal of Swift is to be fault tolerant and it uses replication to achieve this goal. Furthermore, access control lists (ACL) provide fine-grained authorization, and a pluggable authentication framework is integrated in Swift.

4.1.2.4 Conclusion

OpenStack is an active project that has a lively community and a lot of momentum. Nova, the compute cloud service, is currently still a young component. However, it is expanding fast in terms of features, and an aggressive release schedule aims at providing scalability and robustness for large scale deployments in the near future. Swift, the storage cloud, is already production ready and powers RackSpace's commercial storage cloud offering *CloudFiles*, which stores petabyte of data and billions of objects. The development environment is professional and attractive, because of readable code, documentation, active and open developers, a transparent development process, and unit tests. The overall focus of OpenStack is scalability and fault tolerance, therefore opportunities for TClouds exists to have an impact on other security goals such as confidentiality and integrity.

4.1.3 Evaluation and Final Selection

4.1.3.1 Brief Comparison: OpenNebula vs. OpenStack

Comparing the communities of both projects shows that OpenStack features a diverse community consisting of contributors of different organization, whereas OpenNebula is mainly developed by the University of Madrid and the associated spin-off company. OpenStack has a more commercial-driven community, since it is used by RackSpace in their commercial cloud offering and all the supporting organizations are enterprises or startups. OpenNebula originates from a academic and research background, and it is mostly used in other research projects and organizations.

From a technical point of view, there are a number of differences between these two cloud computing platforms. OpenStack aims for a decentralized approach, as manifested by their shared-nothing and message-based architecture, whereas OpenNebula has a more centralized approach (i.e. the OpenNebula daemon). OpenStack is written purely in Python, and OpenNebula's code base consists of mainly C++ code with minor parts written in Java and Ruby. OpenStack is the combination of compute (Nova) and storage (Swift) cloud components, whereas OpenNebula does not provide a storage service. Possibly Swift can also be integrated with OpenNebula. The paradigm of managing multiple clouds with the same platform is only supported by OpenNebula. OpenStack is only able to build isolated single clouds.

4.1.3.2 Final Vote

During the Technical Meeting in Lisbon in M4, after discussing the results of our evaluation, we had no clear winner based on the criteria described in Section 4.1. Hence we decided to vote, with each partner having one vote. Table 4.1 shows the result of the vote.

OpenNebula	OpenStack	Abstain
POL, SRX	IBM, FAU, TUDA	FFCUL, HSR, OXFD, ULD

Table 4.1: Vote on platform choice

We decided to use OpenStack as a starting point for our technical development in the project.

4.2 TClouds Prototype High-level Architecture

In this section we review the initial TClouds architecture specification as detailed in D2.2.1, Chapter 4, and we define the initial prototype architecture as a subset of the overall specification.

Figure 4.5 presents a bird's-eye view of the TClouds architecture. A Trustworthy Cloud

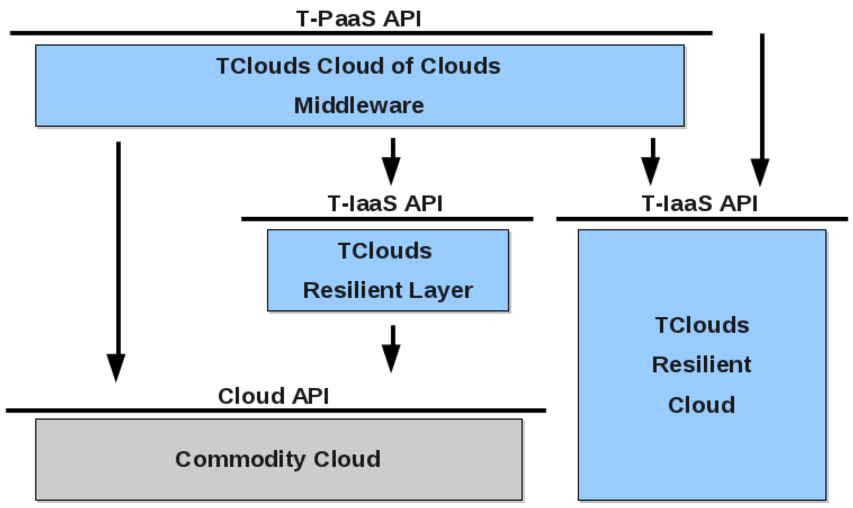


Figure 4.5: TClouds high-level architecture

Infrastructure (focus of WP 2.1) can be realized through a resilient cloud or a resilient layer on top of a commodity cloud. This provides users with Trusted Infrastructure services (T-IaaS). On the other hand, adaptive resilience from cloud of clouds (focus of WP 2.2) can be obtained extending such a T-IaaS API or be built directly on top of commodity clouds. Within the context of cloud of clouds, we discovered that some functionality can not be provided at infrastructure level, but requires the provision of Trusted Platform services (T-PaaS). Another important achievement of Y1 with respect to the initial plans concerns Cross-layer Security and Privacy Management (focus of WP 2.3). We recognized that a central management component is not suitable for our architecture (mainly to allow users and especially providers to pick up only the TClouds functionality they require), therefore we plan to integrate the management directly within each subsystem.

In the following, we provide additional details on the initial architecture specification and we introduce the subsystems that will be developed as part of the integrated prototype. First, in Section 4.2.1, we review deployment alternative for TClouds, showing how it addresses several realistic scenarios of resilient cloud computing and putting in evidence those most relevant for the TClouds prototype. Next, in Section 4.2.2 we recall the key architecture aspects and the main building blocks, providing details on the local architecture of TClouds nodes. Finally, in Section 4.2.3 we introduce the subsystems that will constitute the TClouds prototype and we discuss how they are placed within the TClouds architecture.

4.2.1 TClouds Deployment Alternatives

The final and overall goal of TClouds is to be able to supply a cloud systems architect with as many deployment alternatives as possible.

In particular, our architecture supports implementations of TClouds functionality preserving the use of legacy commodity clouds IaaS, either by resorting to client-side software, or to server-side software. We call them respectively *TClouds Information Agent* (TIA) and *TClouds Information Switch* (TIS), we shall give more details in the following. Moreover, the architecture allows for more ambitious steps, those considering that commodity cloud providers will eventually adhere to a model such as TClouds, directly providing resilient cloud computing.

The rich infrastructure depicted in Figure 4.6 prefigures a true ecosystem capable of offering the best possible tradeoffs to clients and providers of resilient cloud services, either end-clients or mediators.

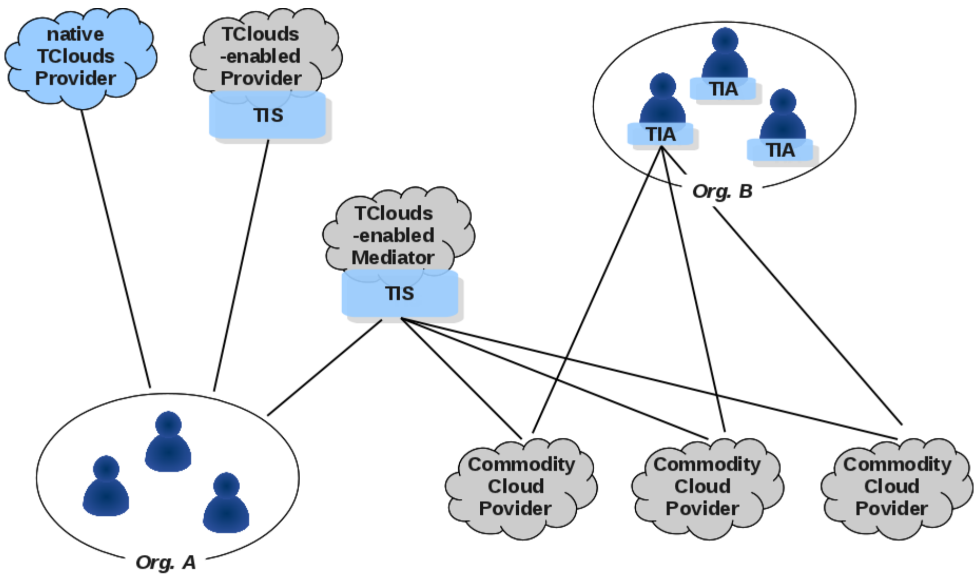


Figure 4.6: TClouds deployment in a diverse ecosystem (source: D2.2.1, Figure 4.7)

From left to right, we can observe end-clients from an organization (Org. A), that access *native TClouds* or *TClouds-enabled* cloud providers (i.e. infrastructures local to a cloud provider, using TCLOUDS protocols and mechanisms). The same clients – that we stress are unmodified – can access resilient cloud services implemented on top of commodity clouds through a *TClouds-enabled mediator*. Finally, another way to achieve resiliency from commodity clouds is via *TClouds-enabled client-resident software*, as shown by users from the other organization (Org. B) in the figure. For more details on these deployment alternatives, including security and dependability problems that arise in each scenario, we refer to D2.2.1, Chapter 4.

Concerning the TClouds prototype which is the focus of this report, the two extreme alternatives will be demonstrated, providing examples of *native TClouds* infrastructures and *TClouds-enabled client-resident software*.

4.2.1.1 Native TClouds Infrastructure

This infrastructure uses native TClouds protocols and mechanisms in the design of the data centers from scratch. This alternative is bound to achieve the product with the highest trustworthiness level and the best performance, that is, ultimately trustworthy IaaS and PaaS for a single cloud provider, but at the loss of diversity.

4.2.1.2 TClouds-enabled Client-resident Software

Client-resident software is composed of add-on modules allowing direct implementation of some secure services over commodity clouds, by organization's clients. This is the simplest

TClouds implementation, not requiring additional machinery, but implying modifications in all client machines wanting to access resilient cloud services. The major point of strength of this solution is the total absence of single points-of-failures.

4.2.2 Initial Architecture Specification

TClouds could be described, in short, as a resilient cloud-of-clouds infrastructure providing automated computing resilience against attacks and accidents, in complement or in addition to commodity clouds. This enhanced functionality will be achieved through specialized middleware standing between low-level, basic multi-cloud untrusted services, and the applications requiring security and dependability.

The TClouds architecture provides applications with a wealth of interfaces to produce incremental resilience solutions with single or multiple clouds: *TClouds Trusted Platform services* (T-PaaS) on top of the middleware layer; *TClouds Trusted Infrastructure services* (T-IaaS) from within the middleware layer; Infrastructure services (IaaS) from available commodity untrusted clouds.

The main building blocks of the architecture that implement this functionality, illustrated in Figure 4.7 (a), are reviewed in the following (for details, we refer to D2.2.1, Chapter 4).

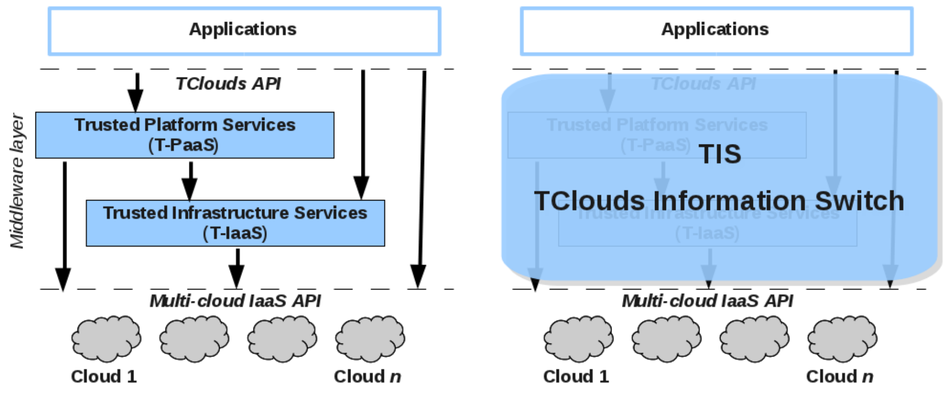


Figure 4.7: TClouds architecture (source D2.2.1, Figure 4.2) with block diagram (left) and TClouds Information Switch (right)

Basic multi-cloud untrusted services represent the available standard functionality, at IaaS level, offered by commodity market players.

Trusted infrastructure services represent trusted-trustworthy versions of IaaS services, namely storage and processing power. The idea is to offer file systems, and low-level virtual machines, resilient to attacks and faults, by combinations of fault/intrusion prevention and tolerance mechanisms and protocols which build a resilience layer on top of the corresponding untrusted storage and processing systems.

Trusted platform services represent trusted-trustworthy services at a higher level of abstraction, provided through extensions of the resilience layer implemented by the TClouds middleware, built on top of either or both the IaaS and the T-IaaS. These services normally deploy a semantics useful to build complex reliable and distributed applications. Examples are: state machine replication, consistent service execution, etc. Once more, these services are implemented by combinations of fault/intrusion prevention and tolerance mechanisms and protocols, for example, Byzantine fault-tolerant (protocols).

In order to support the deployment alternatives described above, we introduce the notions of *TClouds Information Switch* and *TClouds Information Agent*.

TClouds Information Switch (TIS) is a conceptual "box" which runs the middleware protocols and mechanisms implementing the resilience components already mentioned, see also Figure 4.7 (b). Each TIS instantiation encapsulates the services in use by that configuration, which are all or part of the services defined in the TClouds architecture. TIS implementation depends on the particular incarnation, ranging from dedicated machine to fault and intrusion tolerant appliance box containing several TIS replicas implemented as virtual machines. The TIS can be built with incremental levels of resilience, depending on its criticality. Trustworthy TIS-TIS interconnection through TClouds communication services secures information flows in the architecture.

TClouds Information Agent (TIA) can be seen as a particular implementation of a TIS, as a software appliance residing with end clients. Like the TIS, it runs different sets of functions, depending on specific protocols being used on the client side. TIAs require no additional hardware as a general rule. However, running in the client space, they are subject to a great level of threat. This can mitigated by configurations where the TIA logic is aware of the existence of minimal additional hardware (e.g., trusted components) to improve its resilience. On the other hand, the TIA option requires client modifications to achieve the desired TClouds functionality. Whenever needed, trustworthy TIA-TIS interconnection through TClouds communication services secures information flows in the architecture.

We conclude this section with details on the local architecture of TClouds nodes. The description here slightly extends the one in D2.2.1, Chapter 4, but is limited to the cloud node and does not cover other components for adaptive resilience.

A snapshot of the local architecture of TClouds detailing a node and its interconnection methods is depicted in Figure 4.8. This architecture is suitable for TIS but also for nodes in native TClouds data center.

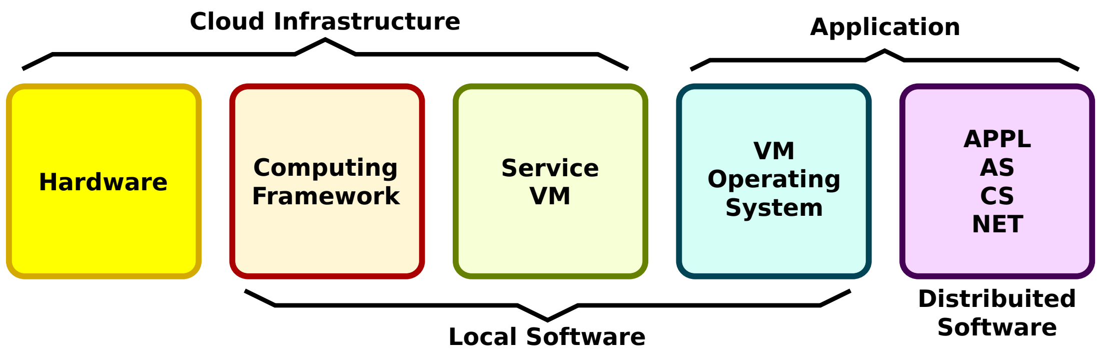


Figure 4.8: Local architecture of a TClouds node (see D2.2.1, Figure 4.8 for comparison)

Firstly, there is the *hardware* dimension, which includes the node and networking devices that make up the physical distributed system. In general, we assume that most of a node's operations run on untrusted hardware, e.g., the usual machinery of a computer, connected through the normal networking infrastructure. However, some nodes – TIS, for example – may have pieces of hardware that are trustworthy, i.e., where by construction intruders do not have direct access to the inside of those components. The types of trustworthy hardware featured in TClouds may include standard TPMs, or dedicated appliance boards with processor, plugged into the node's main hardware.

Secondly, services based on the trustworthy hardware are accessed through the local support services, or *local software*. The rationale behind our trusted components is the following: whilst we let a local node be compromised, we make sure that the trusted component operation is not undermined (crash failure assumption). Within this layer, we can further distinguish between *cloud infrastructure* and *cloud application software*. The former is composed of a *cloud computing framework* (in our prototypes either built upon OpenStack or on SRX Trusted Infrastructures, cf. D2.1.1, Chapter 12) and *Service VMs*, i.e. VMs not dedicated to applications but providing IaaS/PaaS services. The latter consists of the application VM operating system and basic services.

Thirdly, there is the *distributed software* provided by TClouds, i.e. middleware layers on top of which distributed applications run, even in the presence of malicious faults.

4.2.3 Prototype Subsystems

We now introduce the subsystems that shall constitute the TClouds prototype. Each subsystem is shortly introduced here whereas details are provided in dedicated sections in Part II of this report.

4.2.3.1 Trustworthy Cloud Infrastructure (WP 2.1)

Improved Availability and Resilience

- **Resource-efficient BFT (CheapBFT) [FAU].** Services provided via world-spanning networks such as the Internet has been getting more and more important for today's society. This becomes evident, when these services are unavailable or, even worse, when they produce incorrect results. One way to improve the availability and reliability of services is the usage of replication. However, in the standard variant, replication only regards outages caused by crashes. Tolerating arbitrary faults, called "Byzantine faults" such as software bugs, intrusions, viruses, hardware faults etc. and hence ensure correct results is far more difficult and resource intensive. For this reason, industry is reluctant to consider Byzantine faults, since it presumably entails relatively high financial costs.

  Addressing this problem, FAU will contribute with an approach for Byzantine fault tolerance, called *CheapBFT*, exhibiting a lower resource footprint and thus making its usage more practical. Among other techniques, CheapBFT will rely on a trusted hardware component to meet its objectives.

- **Simple Key/Value Store [FAU].** FAU will provide a simple key/value store as an example for a simple cloud service component. This can be used by other services to cache non-critical data, i.e. dynamically generated frontend websites. It can also be used as an minimal example for the downstripped and hardened components we intend to develop in WP 2.1. Once the basic functionality is in place, the service can be extended with authentication mechanisms and different access protocols (memcached, REST, SOAP, ...) to allow better integration and storage of sensitive data.

Root-less Environment

- **Secure Block Storage (SBS) [TUDA].** TUDA will contribute *Secure Block Storage* (SBS). Block storage is non-linear raw memory attached to VM instances as block device (virtual hard disk, e.g. iSCSI). SBS will provide a transparent layer that provides security properties such as *confidentiality, integrity* and *authenticity* for block devices. The SBS is also responsible for user-centric key management.

- **Secure VM Instances [TUDA].** Based on the Secure Block Storage component (SBS), TUDA will contribute with a component that allows clients to securely deploy, launch, and migrate their own VM images. The component ensures that the VM images and data contained within will be confidentiality and integrity protected when they are at rest in a image repository or in transit during migration. The authenticity can be ensured using a secure channel.

- **TrustedServer [SRX].** SRX will provide the TrustedServer as the central security platform to run the VM instances (also called compartments). It is based on the TURAYA Security Kernel and provides isolation of compartments by linking them to TVD s. Domain specific transparent encryption is applied to prohibit information flow between TVDs. The focus of this component is to provide (together with the TrustedObjects Manger (TOM) a trusted platform for cloud applications from the ground up.

Verification and Auditability

- **Log Service [POL].** Log Service is the TClouds logging subsystem, mainly used by other Cloud Components to log their internal events and possibly also used by applications. Log Service can be used as basis for auditing or reporting SLA compliance to the User (here the main target of the service is the end user of the cloud, but it may also refer to an external auditor or to the Cloud Admin). In WP2.1, we concentrate in providing integrity of logs, privacy-aware and access control mechanisms, whilst in WP2.2, we concentrate on availability and logging of cloud of clouds events.

4.2.3.2 Cloud of Clouds Middleware for Adaptive Resilience (WP 2.2)

Improved Availability and Resilience

- **State Machine Replication [FFCUL].** Server and client are the basic structures used to implement distributed systems as clouds. The server offers services and the client use such services by invoking them. An invocation is done by sending a request message from the client to the server, which sends the corresponding results with a reply message to the client.

  Fault-tolerant distributed systems are implemented by replicating the components prone to failures, so they can fail independently without compromising the service availability. An intrusion-tolerant system is commonly modeled as a fault-tolerant system, capable of defending itself against *byzantine failures*, in which a component is allowed to fail in arbitrary ways, including the most common stop and crash failures, but also processing requests incorrectly, corrupting their local state, or producing incorrect or inconsistent outputs.

  Byzantine fault-tolerant services are implemented using *replicated state-machines*, that upon receiving a request deterministically change to a new state and send a reply. All state-machine replicas start with the same state and requests are sent to them using reliable, ordered, broadcasts from clients. Majority (voting) is used in the clients to select the correct reply among those from all replicas. The section 8.1.3 gives more details on how this is actually done.

- **Fault-tolerant Workflow Execution (FT-BPEL) [FAU].** FAU will contribute with a PaaS infrastructure permitting the fault-tolerant execution of business processes in particular and workflows in general which are based on and composed of Web services. The infrastructure will be based on *BPEL*, an XML-based language for describing such workflows.

Resilient Storage

- **Resilient Object Storage [IBM+FFCUL].** The object model for cloud storage has become extremely popular, after its introduction with Amazon's Simple Storage Service (S3) in 2006. It allows reads and writes of simple blobs, each one identified by a unique name (also called a "key"). A multitude of commercial providers offer such *blob storage* services today.

  IBM and FFCUL will contribute a system that builds reliable and secure storage through a federation of object storage services from multiple providers. Multiple clients may concurrently access the same remote storage provider and operate on the same objects. They do this through an interface that contains the basic and most common operations of object cloud storage. (Since every vendor provides the same basic operations but slightly different advanced operations, the system only uses the common denominator of all providers.)

  The software is a library run by each client before it accesses cloud storage; the management and setup is the same as for accessing one storage provider, and the library does not require client-to-client communication. The library requires some cryptographic credentials (public keys) of all clients to be present.

  The storage system provides confidentiality through encryption, integrity through cryptographic data authentication, and reliability through data replication and erasure coding. Key management for encryption and authentication keys is integrated.

- **Confidentiality Proxy for S3 [SRX].** SRX will contribute to the trusted cloud infrastructure with a confidentiality proxy for S3. The component is implemented as a security service which is part of the SecurityKernel and managed by TOM. It will transparently encrypt data of a mounted file system (Linux) according to a TVD and allows to integrate untrusted Amazon Simple Storage Service (Amazon S3 [amab])-based storage into the trusted cloud infrastructure. The S3 proxy does not directly expose the S3 interface to the User. Instead the S3-based storage is mounted as a file system (via s3fs [s3f]). So a User stores and reads ordinary files through a Linux file system or (optional) a Server Message Block (SMB) share instead of accessing the bucket(s) directly, which would mean interaction with buckets and objects via the SOAP and REST API [amac]. The encryption happens transparently within the TrustedServer which attaches the S3-based storage as an encrypted file system to all VM instances belonging to a TVD The encryption key is derived from the TVD f the VM instance and managed by TOM. The main purpose of this component is to demo an integrated prototype: Management, TrustedServer, and untrusted Cloud.

4.2.3.3 Cross-layer Security and Privacy Management (WP 2.3)

High-availability Management

- **Access Control as a Service (ACaaS) [OXFD].** The provision of automated management of Clouds virtual resources is a fundamental requirement for the success of future Cloud. Such automated management would require understanding the properties of Cloud infrastructure and its policies, and it would also require understanding Cloud user requirements. Matching user requirements and infrastructure properties in normal operations as well as during incidents would result in ensuring Cloud requirements are continually considered by Cloud provider. In this part we are planning to develop an Enterprise Rights Management (ERM) tool, which we refer to as Access Control as a Service (ACaaS).

  The objective of ACaaS is to act as a policy decision point to manage the hosting of VM instances at an appropriate Computing Node. Specifically, ACaaS component verifies that a Computing Node satisfies User requirements when hosting its VM instance. This is achieved by matching Cloud's User requirements and Computing Node infrastructure policy/properties.

Secure Management of Keys and VM images

- **TrustedObjects Manager (TOM) [SRX].** The Trusted Objects Manager (TOM) is the central management component of the trusted cloud infrastructure. The TOM manages the physical infrastructure including networks, services and appliances (physical platforms). Since appliances remotely enforce a subset of the overall security policy, a permanent trusted channel between the TOM and its appliances is used for client authentication, to check their software configuration using attestation, and to upload policy changes and software updates. Finally, for each TVD efined the TOM creates an independent TVD-specific Root-CA. SRX will contribute to enhance the TOM to manage the TrustedServers within the cloud infrastructure. TOM manages TVD s and Inter-TVD information flow policies, provides key-management and configures the managed TrustedServers accordingly.

  As the central TVD Management Component of a TVD-based infrastructure TOM provides the user interface to define TVDs and corresponding intra-TVD and inter-TVD information flow policies.

- **Trusted Management Channel [SRX].** The Trusted Management Channel allows to securely connect the TOM with TrustedServers to setup, start and stop VM instances, and to load configuration and policies. It also could be used to interconnect TOMs.

Verification and Auditability

- **Ontology-based Reasoner to Check TVD Isolation [POL].** The ontology-based Reasoner is a subcomponent/plugin for the Management Component that, given as input a service model, an infrastructure model and an allocation of services onto the infrastructure, makes it possible to verify whether some security properties required by the service are satisfied by the allocation. Furthermore, it may also provides hints on how to modify the allocation whenever security requirements are not met.

  More specifically, the service model shall describe a TVD as a virtual network and the main property we shall verify is isolation, in part achieved at "computational" level by the hypervisor, in part achieved at network level by securing untrusted channels. We rely on ontology-based reasoning to perform analysis.

- **Automated Validation of Isolation of Cloud Users [IBM].** *SAVE* (Security Assurance for Virtual Environment) is a tool developed at IBM research for extracting configuration data from multiple virtualization environments, transforming the data into a normalized graph representation, and subsequent analysis of its security properties. IBM will integrate and adapt this technology for the demonstrator based on OpenStack, in order to validate isolation of cloud users.

In Figure 4.9 we show how TClouds subsystems are placed withing the local architecture shown in Figure 4.8. Said in other words, we summarize in a graphical form how, and in which layer, each subsystem shall provide his enhance security or dependability features.

First we distinguish if the subsystem resides at the Cloud Provider side (TIS) or it is designed to be integrated at Client side (TIA), or both. For TIS, we further refine according to Figure 4.8 in the cloud infrastructure (hardware, cloud framework, service VM) and cloud application (VM operating system and basic services, TClouds middleware). In the TIA, we consider the hardware (HW), the operating system and basic services (OS) and the TClouds middleware.

A dark cell in a subsystem row means that the component implements a feature in the corresponding layer. A light cell means a possible extension of the subsystem. As none of our subsystems directly implements hardware solutions, we used that column to specify hardware requirements.

Subsystem	Cloud Provider side (TIS)					Client side (TIA)		
	Cloud infrastructure			Cloud application				
	HW	C. Framework	Service VM	VM OS	Middleware	HW	OS	Middleware
Resource-efficient BFT (CheapBFT) [FAU]	FPGA	■			■			■
Simple Key/Value Store [FAU]			■					
Secure Block Storage (SBS) [TUDA]	TPM	■						
Secure VM Instances [TUDA]	TPM	■	□					
TrustedServer [SRX]	TPM	■						
Log Service [POL]	TPM	■	■		□			□
State Machine Replication [FFCUL]					■			■
Fault-tolerant Workflow Execution [FAU]			■		■			
Resilient Object Storage [IBM+FFCUL]								■
Confidentiality Proxy for S3 [SRX]							■	
Access Control as a Service (ACaaS) [OXFD]	TPM	■	□	■				
TrustedObjects Manager (TOM) [SRX]	TPM	■						
Trusted Management Channel [SRX]	TPM	■						
Ontology-based Reasoner [POL]		■	■					
Automated Validation [IBM]		■	■					

Figure 4.9: TClouds subsystems and their placement within the local architecture

Chapter 5

Preliminary API Definition

Chapter Authors:
Michael Gröne, Norbert Schirmer (SRX)

This chapter introduces the preliminary API of the TClouds platform. It consists of two parts. First, in Section 5.1, we provide an overview of the OpenStack API, Trusted Infrastructures API and Cloud-of-Clouds API as starting points for our technical development. Moreover, in Section 5.2, we classify the APIs of the subsystems that constitute the TClouds prototype according to different relevant parameters (type, functionality, client, deployment, ...).

5.1 Introduction to Application Programming Interface

The goal of an Application Programming Interface (API) is to give a particular set of rules and specifications that services, such as software or infrastructure, can follow to communicate with each other. It serves as an interface between different services and facilitates their interaction, similar to the way the user interface (e.g. GUI) facilitates the interaction between humans and computers.

An API can be created for applications, libraries, etc., as a way for defining their "vocabularies" and resource request conventions (e.g., function-calling conventions). It may include specifications for routines, data structures, object classes, and protocols used to communicate between the consumer program and the program implementing the API. An API can also be the definition of a protocol.

5.1.1 Openstack API

The OpenStack API (cf. [Ope10, Ope11]) includes management functions to be used in the Cloud Infrastructure, built using OpenStack components and relevant for D2.1.1. In that deliverable a security analysis of OpenStack and its API (cf. D2.1.1, Chapter 4) is described.

5.1.1.1 OpenStack REST API

Each core project (Nova, Swift and Glance) exposes one or more RESTful interfaces to interact with the outside world. These RESTful interfaces may be used by the public (Public API), or operators (Management API). Orchestration and higher level systems should also use these APIs. Projects may also expose notification interfaces. These should also be based on the REST principles. The RESTful APIs have a minimum set of standards and capabilities (they will all be versioned, they should all be extensible and support rate limiting, etc., cf. [Ope11]).

5.1.1.2 Dev APIs (Internal APIs)

DevAPIs help developers in building OpenStack components. These APIs are targeted exclusively to project developers. They need not to be RESTful (though they can be), they might leverage other protocols, they may be Python based or whatever. DevAPIs should not be exposed outside a project's boundary. For example, if Nova needs to interact with Swift, it should interact via the OpenStack API, never via the DevAPI, that is restricted to developers (cf. [Ope11]).

5.1.1.3 Design Notes

The Openstack API (cf. [Ope11]) should be a superset of Rackspace APIs:

- Rackspace APIs currently have the goal: "Launch and control Cloud Servers programmatically using a RESTful API"
- OpenStack API has a similar goal: "Launch and control Cloud Systems programmatically using a RESTful API"
- Needs higher levels abstractions for datacenter/Cloud System, composite application and so forth
- Introduces Storage and Network layer abstractions.

5.1.2 Trusted Infrastructures API

As shown in D2.1.1, Chapter 12, Trusted Infrastructures consist of SRX components of which the central management component, the TOM, manages all appliances, such as TrustedServer. The API consists of the GUI of the TOM, the protocol and the libraries for management.

5.1.3 Cloud of Clouds middleware API

Currently, there is no preliminary API targeting the Cloud of Clouds (CoC) middleware approach yet. First approaches to the CoC middleware are shown in D2.2.1, Chapter 5.

5.2 TClouds Subsystems Preliminary API

In this section we give a brief overview of the parts of the TClouds subsystems preliminary API.

5.2.1 Kind of API

TClouds API mostly consists of public subsystems APIs. These may be of the kind:

- GUI
- RESTful (conforming to the REST constraints)
- SOAP
- Protocol (implementation of a protocol)
- Library.

5.2.2 Functionality of API

Functionality, such as:

- Infrastructure hardening
- Compute
- Storage
- Image mgnt
- Extensions.

5.2.3 Clients of API

Clients of the API of the subsystems may be:

- Other Cloud subsystems
- Cloud Admin (GUI)
- Users (GUI)
- Developers.

5.2.4 Deployment of API

Another part of the preliminary API is the information about where subsystems are deployed. The deployments of API are:

- OpenStack
- Trusted Cloud (SRX) / Trusted Infrastructure
- Cloud-of-Clouds.

5.2.5 Standards used or extended

The TClouds API covers a range of standards used by components and subsystems, such as Amazon S3, OpenStack, and introduces extensions of them and new ones:

- Amazon AWS (EC2, S3, etc.)
- OpenStack
- Other standards (e.g. TLS, BPEL, syslog, memcached, ZooKeeper, etc.)
- Extension of a standard
- New one.

5.2.6 Possible Groups of API

Components and their API could be grouped into 4 groups:

- Infrastructure extensions, such as logging, trusted infrastructure, fault tolerance
- Audit components, such as reasoner and automated validation
- Middleware (such as S3 proxy or BPEL)
- Storage (such as key/value/object store).

5.3 TClouds Subsystems Preliminary API Table

The following table shows the TClouds subsystems public APIs.

	Kind	Clients	Deployment	Standard
CheapBFT (FAU)	protocol	Other Cloud subsystems	OpenStack	Smart-BFT
Simple Key/Value Store (FAU)	protocol	Other Cloud subsystems	OpenStack	memcached
Secure Block Storage (TUDA)	REST	Users, other Cloud subsystems	OpenStack	Extension of OpenStack
Secure VM Instances (TUDA)	REST	Other Cloud subsystems	OpenStack	Extension of OpenStack
TrustedServer (SRX)	protocol, libraries	Other Cloud subsystems	Trusted Cloud	Extension of infrastructure
Log Service (POL)	REST, GUI	Other Cloud subsystems	OpenStack	Extension of syslog, Extension of OpenStack
State machine replication (FFCUL)	protocol	Other Cloud subsystems	Cloud-of-Clouds	Smart-BFT
FT-BPEL (FAU)	SOAP, other protocol	Users, other Cloud subsystems	OpenStack, Cloud-of-Clouds	BPEL process, ZooKeeper
Object Storage (IBM+FFCUL)	REST	Other Cloud subsystems	Cloud-of-Clouds	Extension of S3
Confidentiality proxy for S3 (SRX)	GUI, other	Other Cloud subsystems	Trusted Cloud	Linux file system
Access Control as a Service (OXFD)	GUI	Users, Cloud Admin	OpenStack	Extension of OpenStack
TrustedObjects Manager (SRX)	GUI	Other Cloud subsystems, Cloud Admin	Trusted Cloud	Extension of infrastructure
Trusted Management Channel (SRX)	protocol, libraries	Other Cloud subsystems	Trusted Cloud	Extension of infrastructure, Extension of TLS
Ontology-based Reasoner (POL)	GUI, Java	Cloud Admin	OpenStack	New
Automated Validation of Isolation (IBM)	REST	Cloud Admin	OpenStack	New

Table 5.1: TClouds subsystems preliminary API

Chapter 6

Test Methodology

Chapter Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)
Mina Deng (PHI)
Ilaria Baroni, Marco Nalin (HSR)
Paulo Jorge Santos (EFA)

Testing is essential to guarantee the quality of software components. In order to meet the functional requirements and to match the desired quality level, TClouds subsystems will be evaluated as standalone components as well as group of components that cooperate to form the cloud infrastructure.

This chapter introduces basic concepts of software testing and explains how they will be applied to the TClouds design and development.

6.1 Introduction to Software Testing

The goal of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions, however it can evidence those specific conditions that make the software not working properly. Software testing often includes examination of code as well as execution of that code in various environments and conditions.

Even if multiple different techniques for software testing are available, it is possible to group them into two main categories:

- **White-box:** this testing (also known as clear box testing) employs the knowledge of the internals of the software being tested to exercise different paths within the module, loop boundaries and data structures.
- **Black-box:** in this testing approach, there is no knowledge about the internal structure of the component to be tested. Here, test cases are built around specifications and requirements, i.e., what the application is supposed to do.

In both the cases, the tester chooses inputs to stimulate code and determine the appropriate outputs. Then the inputs are fed to the component to be tested and the outputs are examined. If the obtained output are equal to the expected ones, the test is passed, otherwise the test fails.

White-box and black-box testing can be used to test different aspects of software components:

- **Unit test:** these tests verify if the software units (i.e., the smallest software elements identifiable into a software component) work properly. In practice, the programmer can write unit tests to verify that the methods (or functions) of his program are correct. These tests make it possible to identify bugs in the implementation or errors in the logic of the single components. These test can only be white-box since it is necessary to have a deep knowledge of the internal code for testing all the possible code branch or the loop boundaries.
- **Integration test:** these tests are used to verify how a single component works when put together with other components. Usually these tests identify errors across the interfaces of the components. These tests can be both white-box or black-box and usually are performed by the programmers of the components.
- **Functional test:** these tests aims to ensure that the functionality specified in the requirement specification works.
- **System test:** these tests are similar to functional tests, but they try to verify how TClouds subsystems work when they are composed together into the cloud and act as a single integrated system. Functional and system tests are usually black-box and may help in finding incorrect or missing functionality, interface errors or errors in data structures used by interfaces, and behavior or performance errors.
- **Regression test:** these tests are run through all the other testing phases and verify that modifications to components (which may pass single unit tests) do not have side effects on the rest of the system. Regression test can be thought as a spot-examination of the whole system which is run often. Any strange result of the regression test may indicate that there are problems due to recent modifications.

  Since it is impractical to run all the tests too often, in practice regression tests are a subset of the original test cases which can be completely automated.

6.2 Component tests

In TClouds, each partner may release a testsuite together with its components. This testsuite, if completely automated, can be used, from the perspective of the TClouds project, as unit testing for that component. Whenever possible, components testsuites can be integrated in a single testing framework, for instance Hudson [hud] (also used by the OpenNebula project), so to makes it possible a *continuous integration* test methodology.

6.3 API tests

Internal and TClouds API tests are similar, while they apply to two distinct interfaces. Internal API tests are functional tests for TClouds subsystems, while TClouds API tests correspond to integration tests of the TClouds subsystems with applications, both performed as black-box tests. The internal API tests aim to ensuring that the functionality specified in the requirement specification works. Moreover, this test may involve putting the subsystem in many different environments to ensure it works under different conditions. The TClouds API tests verify how TClouds subsystems work when they are composed together into the cloud and act as a single integrated system.

The API tests should cover the totality of the functions exposed by the subsystems, at least for one fixed condition (i.e., function parameters) and for the default platform configuration. These tests will be automatic, e.g. with REST interface and Selenium [sel], with the possible exception of the platform configuration that may require manual intervention to be changed.

6.4 Application tests

Application tests correspond to the tests of the two prototype applications, that we shall develop in Activity 3 to demonstrate the functionality of TClouds. These tests should be as automatic as possible, e.g. by using Selenium or SIKULI [sik], and must cover at least the interactions with the infrastructure, through the TClouds API, and the graphical user interface.

However, some tests cannot be performed automatically, but require an human to check whether data provided are handled correctly by the application and the latter returns the desired results.

A common set of criteria that should be taken in account during the tests definition are the performance and the resources usage. The first is important because applications are expected to process data in real-time and should not be affected by bottlenecks. The second because an intensive usage of CPU, memory or storage media may severely impact in the costs required to run the application in the cloud. Developers should verify that their applications have good performances and low requirements in terms of resources usage using existing techniques, like the *software profiling*. Software profiling is a program optimization task that runs a performance analysis tool called a profiler with an application under study. A profile of the program's dynamic behavior under a variety of inputs is presented by the profiler and represents the program's behavior from invocation to termination. Profiling is a form of dynamic program analysis that measures, for example, the usage of memory, the usage of particular instructions, or frequency and duration of function calls. Modifications to the programmer's code or to the compiler's settings can serve as the experimental variable in an effort to increase efficiency (speed or memory requirements) of the program under study.

6.4.1 Application Test Plan

Applications will be tested on three different levels. First the internal behavior of the components that belong to the applications is tested. Then, the interaction among components is evaluated. Finally, a typical usage of the application is perfomed. This last test are particularly interesting since they triggers interactions among many application components and, consequently, they also involves many interaction among TClouds internal components. The next table present a preliminary test plan for the two applications from Activity 3 (we refer to D3.1.1, Chapter 5, and to D3.2.2, Chapter 4 for the architecture of the two applications).

6.4.2 Details on Testing TClouds Healthcare Scenario

At a very basic level, in the TClouds Healthcare scanario, a patient health related information must be retrieved from a Philips Actiwatch and sent to the cloud for further processing. A Philips Actiwatch is a wrist-worn hardware device which captures various health related metrics such as patient activity and environment lighting. The ActiWatch that is used in this proof-of-concept is the **ActiWatch Spectrum** (Figure 6.1), which is the most functional device from the Philips Respironics ActiWatch family [?]. It can capture data at 60, 30 or 15 seconds intervals.

	Home Healthcare	**Smart Lighting System**
Component	PHR Service: None H&W Service: None	SL Server: None SL Gateway: None
Components interaction	PHR – H&W: Manual PHR Service – TClouds: Manual (Firefox REST client)	SL Server – SL Gateway: None
GUI	Web GUI: Manual	Web GUI: Manual

Table 6.1: Test plan for applications from Activity 3

Figure 6.1: Healthcare scenario: ActiWatch

The simplified data flow of the Healthcare scenario is shown in Figure 6.2. For a more complete description of the application, refer to D3.1.1, Chapter 5.

6.4.2.1 Testing functionality

Informally, the following workflow is very suited for testing the software:

1. Start the middle-tier and front-end services. Start up the client software.
2. On the client, drop one of the `.csv` files from the *Useful Auxiliaries* into the watch directory.
3. Verify that the client application detects the new file and starts uploading.
4. Verify that the client completes this upload without any exceptions.
5. With a browser, view the front-end page.
6. Log into the front-end with your credentials and verify that no exceptions were encountered (if there were, the page displays this).
7. Select the week for the data that was uploaded. Verify that the “loading...” image is displayed.
8. Verify that the loading image is interchanged for the graph that you want to see.
9. If there are more graphs in this week-range, verify that they all have been generated.

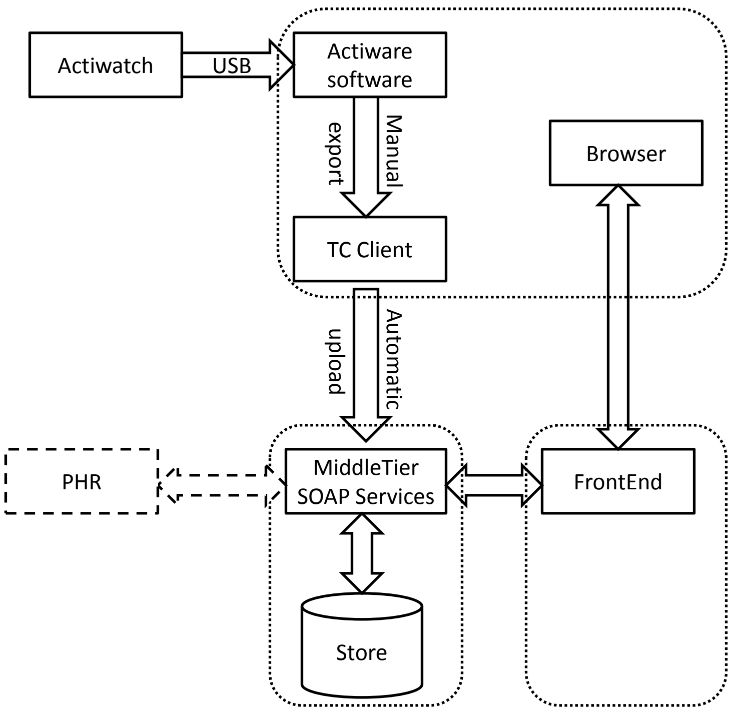


Figure 6.2: Healthcare scenario: simplified data flow

10. Verify that refreshing the page and looking at the graphs again now loads the graphs instantly from the cache.

11. Use the data inspector (drag a frame in an image) and verify that it pops up a new window with the correct data.

Even though unit testing is a common practice in software engineering, we do not use it for this project because the effort involved in unit testing costs us more than we can gain.

6.4.2.2 Testing performance

We applied profiling extensively in this application to measure performance. To do this, we used a Python `profile` module which exports profiling data to a file. We collected this data for all our components. This profiling data is raw and needs conversion. For this we use our `dot.sh` and `gprof2dot.py` scripts. The output of these are *dot* files, which we feed to the `dot` application to generate PNGs of the trace calls.

An example is the retrieving of recordings from the database. See Figure 6.3. What you see here is as follows. The top-most line gives the name of the method involved. The second line indicates the percentage of the whole computing time that is taken up by this function. Between brackets in the third line you find the percentage that is actually taken up by this specific function (as compared to its total time which also takes into account the function calls in levels lower). On the fourth line you see the invocation count showing how many times thus function has been called.

Furthermore, you also see outgoing arrows to other functions calls. These work intuitively and on the arrows you see percentage taken up as well as invocation count. The colors on the method blocks are merely a helpful feature that gives the most redish color to the most CPU intensive methods. Note that this graph is indicative of relative time only, and not absolute time. So one can only learn anything from these graphs by looking at which functions take up the most time relative to the entire trace.

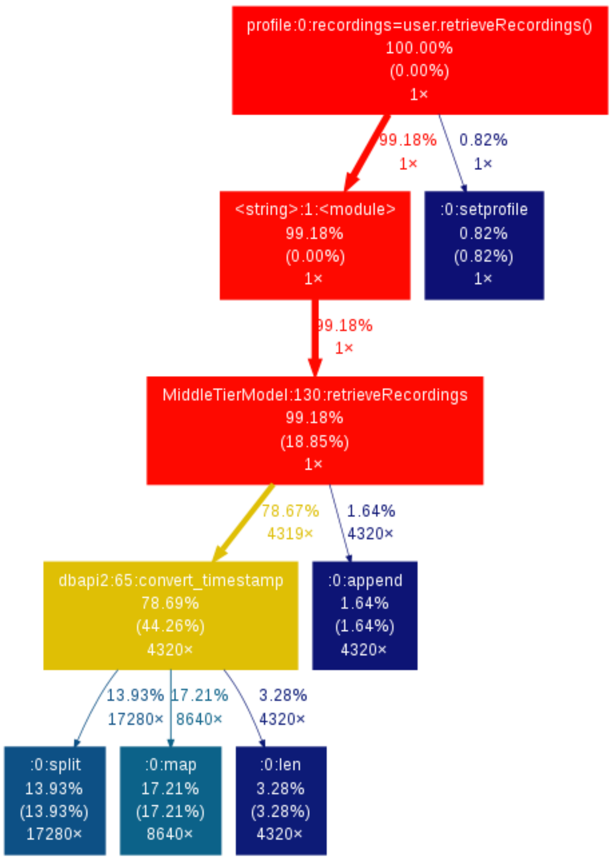


Figure 6.3: Healthcare scenario: calltrace for retrieving recordings

6.5 TClouds Subsystems Test Plan

The tables 6.2 and 6.3 show the tests methodology for each component of the TClouds subsystem and contain the software required for the tests execution when applicable.

	Component	**Internal API**
CheapBFT (FAU)	Client library Replica library	N/A N/A
Simple Key/Value Store (FAU)	Generic Runtime: auto/manual Key/Value store: auto	Local config: manual none
Secure Block Storage (TUDA)	OpenStack Extensions NOVA Extensions Secure Block Storage	verbose logging & pyUnit verbose logging CppUnit, Valgrind
Secure VM Instances (TUDA)		
TrustedServer (SRX)	TMC: manual (TOM)	Not applicable
Log Service (POL)	`libsklog`: Auto (CMockery) Syslog Module for `libsklog`: None	Enhanced Logging in OS Nova: Auto (PyUnit)
State machine replication (FFCUL)	Client library Replica library	N/A N/A
FT-BPEL (FAU)	`output proxy`, `input proxy`, `transformator`: Auto (JUnit) `BPEL engine`, `Web services`: Integrated tests only	Java bindings: Auto (JUnit)
Object Storage (IBM+FFCUL)	Client library	N/A
Confidentiality proxy for S3 (SRX)	StorageProxy: auto (TURAYA Manager)	file system interface (eCryptfs): auto (TURAYA Manager)
Access Control as a Service (OXFD)	VerifyComputingNode(ComputingNode)	N/A
TrustedObjects Manager (SRX)	TMC: manual GUI: manual	Sys parts: none
Trusted Management Channel (SRX)	TMC binary protocol: manual/auto	Not applicable
Ontology-based Reasoner (POL)	Reasoner: jUnit `libvirt` module for Sec Tunnels: Auto (CMockery)	Reasoner API (Java): jUnit Enforcer API (libvirt): manual
Automated Validation of Isolation (IBM)	Discovery: JUnit and manual tests Analysis: JUnit and test data set	N/A

Table 6.2: TClouds subsystems test plan (Component and Internal API)

	TClouds API	Application/User iface
CheapBFT (FAU)	N/A Java bindings: Auto (JUnit)	Java bindings: Auto (JUnit) N/A
Simple Key/Value Store (FAU)	none none	Remote config: auto Usage: auto
Secure Block Storage (TUDA)	protocol:manual N/A Key Provisioning: Depends on Client language (e.g. jUnit)	N/A N/A Small test group
Secure VM Instances (TUDA)		
TrustedServer (SRX)	Not applicable	Mgmt Interface: manual (TOM)
Log Service (POL)	Log Core REST API: Auto (Selenium) Log Storage REST API: Auto (Selenium) Log Storage Syslog API: Manual	Log Console: Manual
State machine replication (FFCUL)	N/A `join`, `leave`, deployment: specific test battery	`invoke`: automated test battery `execute`: client test battery
FT-BPEL (FAU)	Auto (JUnit)	N/A
Object Storage (IBM+FFCUL)	N/A	automated test battery
Confidentiality proxy for S3 (SRX)	file system interface (SMB, s3fs): manual	Mgmt Interface: manual (TOM)
Access Control as a Service (OXFD)	N/A	Mgmt Interface: manual (GUI)
TrustedObjects Manager (SRX)	TMC: manual	Mgmt Interface: manual (GUI)
Trusted Management Channel (SRX)	protocol: manual (TrustedServer)	Mgmt Interface: manual (TOM)
Ontology-based Reasoner (POL)	N/A	GUI: Manual
Automated Validation of Isolation (IBM)	N/A	test scripts

Table 6.3: TClouds subsystems test plan (TClouds API and Application/User iface)

Part II

Selected Subsystems

Chapter 7

Trustworthy Cloud Infrastructure (WP 2.1)

Improved Availability and Resilience

7.1 Resource-efficient BFT (CheapBFT)

Authors:
Johannes Behl, Klaus Stengel (FAU)

7.1.1 Overview

7.1.1.1 Description

Services provided via world-spanning networks such as the Internet has been getting more and more important for today's society. This becomes evident, when these services are unavailable or, even worse, when they produce incorrect results. One way to improve the availability and reliability of services is the usage of replication. However, in the standard variant, replication only regards outages caused by crashes. Tolerating arbitrary faults, called "Byzantine faults" such as software bugs, intrusions, viruses, hardware faults etc. and hence ensure correct results is far more difficult and resource intensive. For this reason, industry is reluctant to consider Byzantine faults, since it presumably entails relatively high financial costs.

Addressing this problem, FAU will contribute with an approach for Byzantine fault tolerance, called *CheapBFT*, exhibiting a lower resource footprint and thus making its usage more practical. Among other techniques, CheapBFT will rely on a trusted hardware component to meet its objectives.

7.1.1.2 Goals (Security, Privacy, Resilience)

- Resilience of services hosted in a trusted cloud

  Description: CheapBFT will be designed in order to ensure availability, reliability and integrity of services hosted in a trusted cloud even in the presence of arbitrary faults. It will be a Byzantine fault-tolerant variant of CheapPAXOS (proposed by Lamport) and will require a trusted hardware module that implements trusted signed counters (see related work form FFCUL, e. g. MINBFT and EBAWA).

  Techniques/research problems: Normally, Byzantine fault-tolerant state machine replication requires $3f + 1$ replicas. CheapBFT will use only $f + 1$ replicas which are actively involved in the execution state as well as in the agreement stage during error free operation. If any kind of misbehavior is detected or suspected, f further replicas will be activated rapidly. Together with a trusted hardware module, the resulting $2f + 1$ replicas are suffice to tolerate f faulty members among them.

Assumptions: Services using CheapBFT have to be deterministic and platforms contain and provide a special trusted hardware module.

7.1.1.3 Required External Components

An FPGA is required as trusted hardware module to implement trusted counters.

7.1.1.4 Relationship with Activity3

To a limited extend, A3 can use CheapBFT to implement Byzantine fault-tolerant services. However, due to a finite number of trusted counters, using CheapBFT for internal services of a trusted cloud seems to be more natural.

7.1.2 Requirements

In order to demonstrate how CheapBFT behaves in practice two byzantine fault-tolerant versions of Apache ZooKeeper[1] are implemented: one based on a customary consensus protocol and one on the basis of CheapBFT. These two implementations and the standard implementation of ZooKeeper, whose underlying Zab protocol tolerates crash-stop failures only, are then compared in terms of performance and resource demand.

7.1.2.1 Selected Use Cases

[1] http://zookeeper.apache.org/

USE CASE UNIQUE ID	**/UC 110/ (Start VM instances)**
DESCRIPTION	A User starts a number of VM instances which will execute the ZooKeeper service. In doing so VM instances have to be only created on Computing Node which possess the hardware module needed by CheapBFT.
ACTORS	User and Computing Node $X_1 \ldots X_f$.
PRECONDITIONS	VM instances were created (cf. /UC 10/) on Computing Node equipped with the trusted hardware module needed by CheapBFT.
POSTCONDITIONS	The VM instances are running on Computing Node $X_1 \ldots X_f$ and their IP addresses are known.
NORMAL FLOW	1. The User starts the VM instances based on a image which contains all three implementations of ZooKeeper (standard, customary protocol, CheapBFT). 2. Since the IP addresses of the instances are needed within the set-up stage, beforehand configured addresses are assigned to the started VM instances. 3. All VM instances are started and are prepared to execute the ZooKeeper service.
ALTERNATIVE FLOW (OTHER WAYS TO CONNECT STARTED VM INSTANCES)	1. Instead of configuring the IP addresses of the VM instances beforehand, they could be dynamically assigned while the starting procedure. In this case, some kind of mechanism has to be provided which can be used to obtain the IP addresses of the started instances.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Stop, reboot or terminate one or more VM instance.

USE CASE UNIQUE ID	**/UC 120/ (Set up and start ZooKeeper service)**
DESCRIPTION	The User sets up and starts a ZooKeeper service.
ACTORS	User and Computing Node $X_1 \dots X_f$.
PRECONDITIONS	VM instances are running on Computing Node equipped with the trusted hardware module needed by CheapBFT.
POSTCONDITIONS	The Zookeeper service is up and running.
NORMAL FLOW	1. One VM instance, let's assume Computing Node X_1, is selected to act as coordinator. 2. It is ensured that the coordinator instance has access to a list containing the IP addresses of all other VM instances (e. g. by means of a simple text file). 3. Set-up scripts are executed on the coordinator in order to initialize all other VM instances (running at Computing Node X_2 to X_f) and to start the desired implementation of the ZooKeeper service (standard, customary protocol, CheapBFT).

USE CASE UNIQUE ID	**/UC 130/ (Retrieve the number of active replicas)**
DESCRIPTION	A user retrieves the numbers of all active replicas.
ACTORS	User and Computing Nodes $X_1 \dots X_f$.
PRECONDITIONS	The ZooKeeper service was set up and is running.
POSTCONDITIONS	The number of active replicas is available.
NORMAL FLOW	1. The coordinator instance is used to get the number of all replicas currently active. 2. The coordinator connects to the other VM instances and collects their current status. 3. The number of active replicas are presented to the user.

USE CASE UNIQUE ID	/UC 140/ (Performing Benchmark)
DESCRIPTION	The User runs a benchmark on the fault-tolerant Zookeeper service.
ACTORS	User and Computing Node $X_1 \ldots X_f$.
PRECONDITIONS	A ZooKeeper service was set up and is running. (/UC 120/)
POSTCONDITIONS	Benchmark results are available.
NORMAL FLOW	1. The User starts the benchmark on the coordinator (Computing Node X_1, cf. /UC 120/). 2. During the benchmark a predetermined set of operations is performed on the selected implementation of the ZooKeeper service (executed by Computing Node X_2 to X_f) and a set of measurements is taken such as elapsed time, transmitted messages, memory demand etc. 3. The results of the benchmark are presented to the User.

7.1.2.2 Demo Storyboard

First a standard Zookeeper installation using the original Zab protocol is started on a set of virtual machines in the cloud and a benchmark is used to measure its performance and resource demand. Then the original ZooKeeper implementation is replaced with a byzantine fault-tolerant version based on a customary consensus protocol and the benchmark is conducted against this version. Finally, the same benchmark is carried out against the CheapBFT version and the results of all three runs are compared.

7.1.3 Architecture

7.1.3.1 High-level Design

CheapBFT provides an efficient consensus protocol which is intended to be used as basis for replicated, byzantine fault-tolerant systems. The efficiency gain compared to customary implementations of consensus protocols is achieved by means of a small trusted hardware module and other strategies to reduce the number of active replicas involved in the process as well as the number of transmitted messages.

Figure 7.1 gives an overview of the architecture of CheapBFT. As stated before, every replica requires access to a trusted hardware module (TSS) implementing trusted signed counters. These counters are used during the protocol executed between the replicas to reach agreement over certain states, for instance the order in which requests of clients are processed.

For that purpose, the trusted modules have to be initialized with a secret key shared by all replicas and used to authenticate messages. Moreover, the modules can be uniquely identified and counters can only be modified from the modules themselves. This way, the counters allow to prevent equivocation, that is, the ability of a faulty or malicious party to make conflicting statements for a single action in the protocol. Without the possibility of equivocation only $2f + 1$ replicas instead of $3f + 1$ are required to tolerate f arbitrary faults.

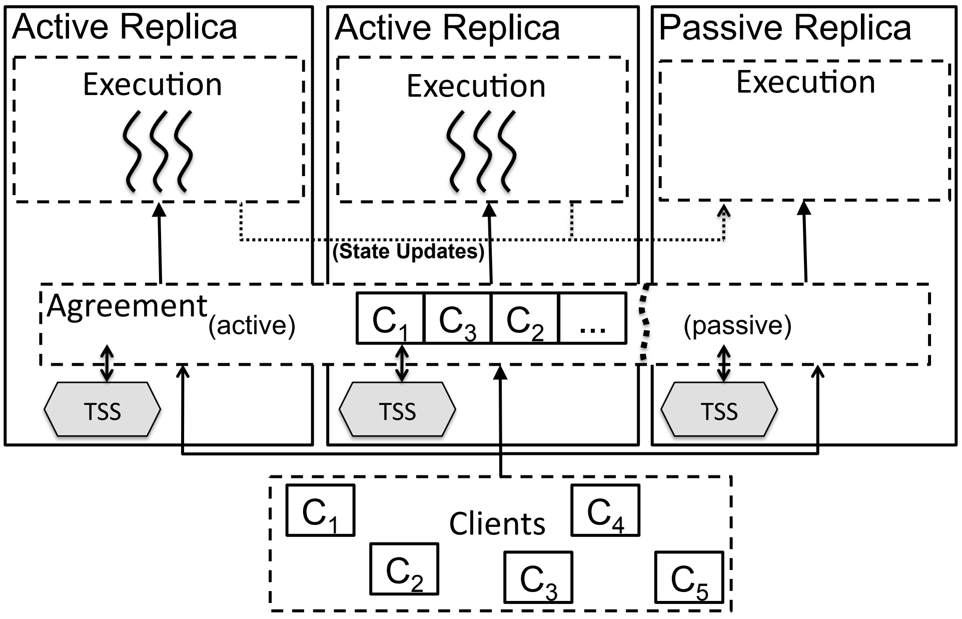


Figure 7.1: A minimal configuration of CheapBFT consisting of two active and one passive replica.

To further reduce the resource demand, CheapBFT distinguishes between *active* and *passive replicas*. If a replica is passive, it doesn't participate in the agreement and execution phases. Instead, they are regularly updated with state changes from the active members, which supposedly requires less resources than executing client requests actively. Since only $f + 1$ replicas are needed to detect errors, f replicas out of $2f + 1$ can stay passive. Here, the state updates ensures, that at the occurrence of errors passive replicas can be turned into active ones without greater delay.

The partitioning of the replicas in $f + 1$ active and f passive ones can be used for another optimization. By dynamically mixing the sets of active and passive replicas, load caused by the agreement and execution stage can be distributed over all replicas.

7.1.3.2 Sequence Diagrams

The sequence executed for setting up a CheapBFT system is depicted in Figure 7.2: The system administrator starts the initialization by providing all replicas with the necessary information such as the connection endpoints of the replicas. After that, the replicas connect to each other and exchange further initial data required to shared starting point all replicas agree on.

Figure 7.3 shows the processing of a request under normal, that is error-free circumstances: A user or client sends a request to all, in this example 3, replicas. However, only the two active replicas execute the agreement protocol, process the request and reply to the user. The third, the passive replica gets only the state changes caused by the request.

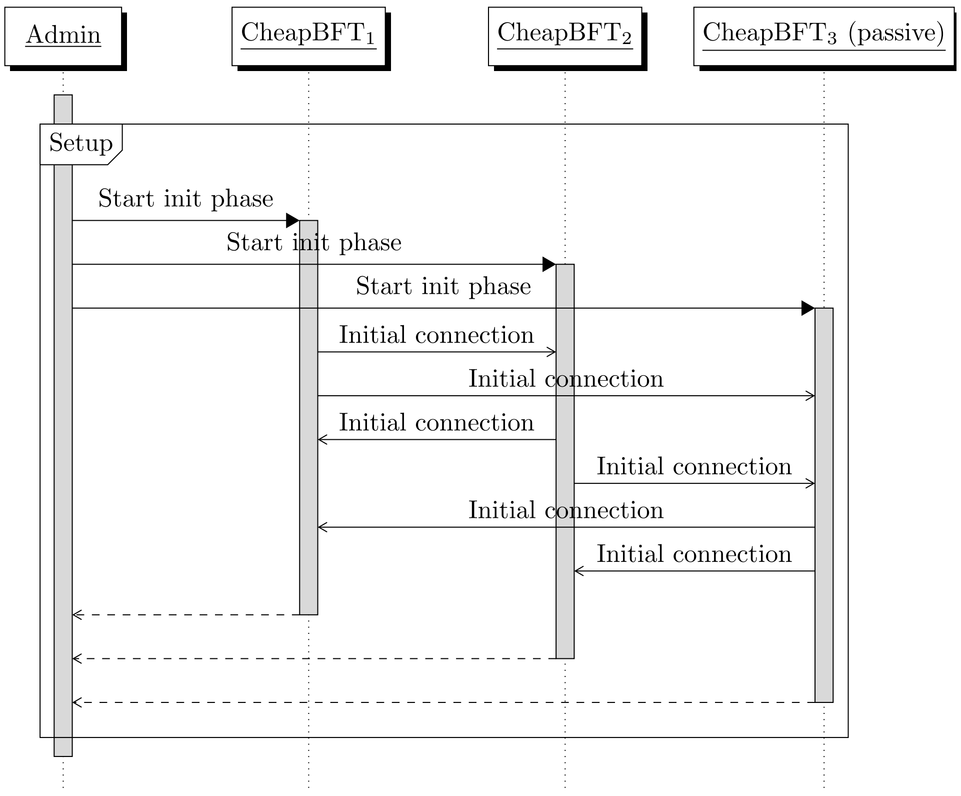


Figure 7.2: Setting up CheapBFT

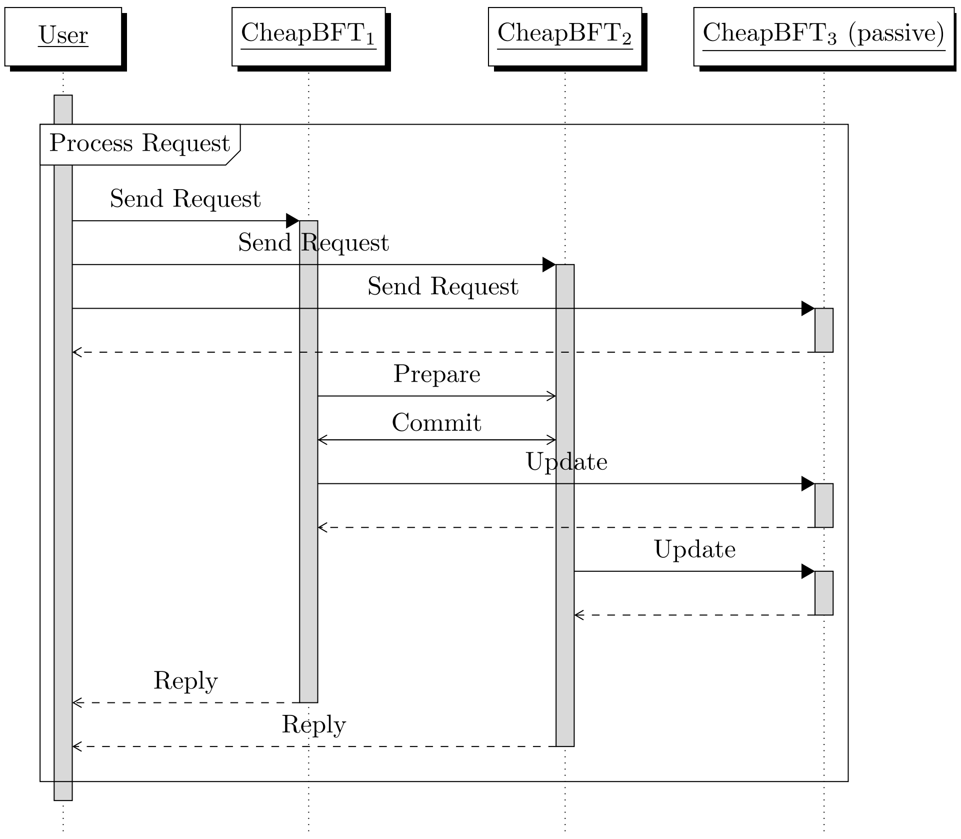


Figure 7.3: Processing of a request by CheapBFT

7.1.4 API

The API of CheapBFT is similar to the API of FFCUL's subsystem presented in section 8.1.

`CreateProxy()`

```
ServiceProxy ⇐ ServiceProxy (int processId, String configHome)
```

Description.
The client creates a proxy object, passing a unique process id and a path pointing to a directory the configuration files.

`InvokeService()`

```
byte[] ⇐ invoke (byte[] command, boolean readOnly)
```

Description.
When the proxy has been initialized, the client can invoke the service by passing a command and a indicator whether this command doesn't lead to state changes.

`CreateReplica()`

```
ServiceReplicas ⇐ ServiceReplica (int processId, String configHome)
```

Description.
Services which are based on CheapBFT have to inherit from an abstract class providing the same information as the client when creating the proxy object:

So far, the API doesn't differ from the API implemented by FFCUL's subsystem. However, realizing a service on the basis of CheapBFT requires a slightly different API:

`ExecuteCommand()`

```
CmdResult ⇐ execute (long timestamp, byte[] nonces, byte[] command, Context ctx)
```

`GetServiceState()`

```
ServiceState ⇐ getState ()
```

`SetServiceState()`

```
void ⇐ setState (ServiceState state)
```

`UpdateServiceState()`

```
void ⇐ updateState (StateUpdate upd)
```

Description.
Although there are also methods for executing commands and getting and setting the whole service state, some additional steps are needed in the case of CheapBFT. To update the state of passive replicas, the execute command must return not only the actual result of the operation but also the state changes. Furthermore, services have to implement a method which adjusts the state of passive replicas according to the state changes returned by the active replicas.

7.2 Simple Key/Value Store

Authors:
Johannes Behl, Klaus Stengel (FAU)

7.2.1 Overview

7.2.1.1 Description

FAU will provide a simple key/value store as an example for a simple cloud service component. This can be used by other services to cache non-critical data, i.e. dynamically generated frontend websites. It can also be used as an minimal example for the downstripped and hardened components we intend to develop in WP 2.1. Once the basic functionality is in place, the service can be extended with authentication mechanisms and different access protocols (memcached, REST, SOAP, ...) to allow better integration and storage of sensitive data.

7.2.1.2 Goals (security/privacy/resilience)

- Integrity of data

  Description: The service and stored data should be safe from unintended modifications.

  Techniques/research problems: We try to achieve these goals by (static) language level checks and runtime integrity checks. Additionally the amount of code running in the system should be minimized. Open Research question is, how to integrate both techniques and find the required lower level operating system parts for a given application scenario.

  Assumptions: Corrupted data or failure of the service is a problem for the application using it.

- Avoid unauthorized access

  Description: In order to store sensitive data or to prevent unauthorized modifications from outside an additional authentication scheme is probably necessary.

  Techniques/research problems: Provide simple enough access control scheme for storage services.

  Assumptions: The application may want to store sensitive data in the cache and needs additional measures to prevent unauthorized access.

- Improved resilience

  Description: It should be possible to improve the resilience of the service for mission-critical data.

  Techniques/research problems: Generate necessary extensions for service replication and data distribution from application description/annotations.

  Assumptions: The application may want to store important data in the provided cloud component.

7.2.1.3 Required External Components

- x86 Virtual Machine

  Description: Requires a secure Intel x86 virtual machine environment as basis for the implementation.

  Features (security/privacy/resiliency): Should provide isolation from other VMs on the same host. Paravirtualized networking and disk access would be nice to improve performance and make development of low-level components easier.

  Required API (provided by the external component): x86 ABI

7.2.1.4 Relationship with Activity3

The key/value store can be used by Activity 3 to accelerate rendering for Web pages and save temporary data.

7.2.2 Requirements

7.2.2.1 Selected Use Cases

7.2.2.1.1 Terminology

- *Generic VM image*: A VM image containing a generic version of our key/value store implementation and operating system stack. It can be started in most Intel x86 virtual machine environment commonly provided by infrastructure clouds.

- *Key/value store*: The specialized operating system and application stack, that was adapted to the specific virtual machine environment and needs of the application that will use the key/value store.

7.2.2.1.2 Actors

- *Web service application*: A web-based service application that was designed to use the key/value store to accelerate recurrent operations. It may run on the same host as the virtual machine containing the key/value store, or alternatively on some other physical machine that is located close to the computer hosting the key/value store.

USE CASE UNIQUE ID	**/UC 150/ (Instantiation of tailored key/value store)**
DESCRIPTION	In this use case, the Cloud Admin sets up the key/-value store and a web service application to use it in order to cache results of some expensive operation. The details regarding the instantiation of individual virtual machines works according to the description in /UC 10/.
ACTORS	Cloud Admin Computing Node X, Y Web service application using the key/value store
PRECONDITIONS	A generic VM image of the key/value store was created at Computing Node X (cf. /UC 10/. An instance containing the web service application is running at Computing Node Y (cf. /UC 20/). The web service application itself wasn't started yet.
POSTCONDITIONS	The web service application is up and running and is ready to use the key/value store component to accelerate recurrent computations.
NORMAL FLOW	1. The Cloud Admin starts the web service application at Computing Node Y. 2. When the web service application starts, it starts the instance of the generic VM image containing the key/value store component at Computing Node X and configures the instance parameters to provide enough resources for the typical usage pattern. 3. When the generic VM image has finished booting, the web service application will be notified and start to specify additional parameters to the generic VM, which describe the expected usage pattern, e.g. which transport protocols are supported and how many key/value pair are going to be stored. 4. The instance of the generic VM image finally collects information about the virtual machine environment and uses it together with the expected usage profile gathered from the previous step to form the tailored system. 5. Once the tailored system is compiled, the generic image will replace itself with the image of the tailored system containing the actual key/value store. 6. The web application now connects to the actual key/value store in order to be able to cache computation results.

USE CASE UNIQUE ID	**/UC 160/ (Run benchmark test for the key/value store)**
DESCRIPTION	The web service application is used to test the performance of the key/value store implementation.
ACTORS	Computing Node X running the key/value store implementation Computing Node Y running the Web service application
PRECONDITIONS	The web service application is up and running and connected to the tailored system with the key/value store.
POSTCONDITIONS	Benchmark results are available.
NORMAL FLOW	1. The web service application determines the current time (wall clock) and saves it for future reference. 2. A predetermined set of read and write operations is performed on the implementation of the key/value store 3. After each operation, the current time is recorded to determine the latency and throughput of the key/value store 4. The resource usage on the key/value store implementation is monitored

7.2.2.2 Demo Storyboard

In the first step, the example web service application is first configured to use a traditional implementation of the memcached. The benchmark in from use case /UC 160/ is run and the results are collected. Afterwards, FAU's tailored key/value store component is set up as outlined in use case /UC 150/ and the benchmark is executed again. Parallel to the benchmark, some slides with the development model are shown to explain some of the security advantages of our approach. Finally the results of the benchmark runs are compared.

7.2.3 Architecture

7.2.3.1 High-level Design

The general architecture is based on a classic client/server model. The only noticeable difference concerns the initialization of the server: In the traditional model, the server that provides services to the client (i.e. other Web application for our demo use case) is typically available when the virtual machine instance hosting it starts. In our model we only start a intermediary service that gathers information about the usage scenario first, so that we can tailor and harden the service componend before we actually start it. Otherwise it behaves almost exactly like any other network service hosted inside a VM. The tailoring process is generic enough, so that it can be adopted to other services than just key/value stores. The key/value store just serves as a minimal useful example how such services can be implemented.

7.2.3.2 Sequence Diagrams

The sequence diagram in Figure 7.4 shows the control flow required to create a new instance of a service. In the first step, the Cloud Admin triggers the start of a VM image containing the generic hardened service component. After a short initialization phase (2), in which a generic runtime environment is booted, the component notifies the application that wants to use the service that it is ready to receive the configuration for tailoring process (3). In step 4, the application sets service-specific properties that describe how the application is going to use the service. Typical examples for such properties are the kind of authentication that should be supported, if any encryption should be, or what specific protocols can be used to access the service. When the set of required properties is transmitted, the runtime confirms that the configuration is valid (5), or aborts the process here in case of conflicting requirements. After the application requrements are determined, the generic runtime of the hardened component gathers information about the Hypervisor it is running on (6 and 7). Now we have all data that is required for the tailoring process, which is finally started in step 8. When the process has finished, the hardened service component replaces the generic runtime system and notifies the application in step 9, that it is ready to be used.

Finally steps 10 and 11 illustrate the typical Request/Response usage pattern for the hardened component. Each time the application needs to interact with the service, it sends an request to the hardened component, which is anwering it eventually. We do not intend to support operations that are initiated by the hardened component itself, nor requests that don't, at least, issue any confirmation message as a reply.

7.2.3.3 Low-level Design

Language

First implementation on small Java system, later ported to safe static language (e.g. ATS/Haskell/...) and annotations for improved run-time integrity checks and simplified replication.

7.2.4 API

Public or private? All APIs are public. The cloud user, which can also be a cloud administrator, has the option to restrict access to the management APIs to internal managment infrastructure.

New or extended? The basic API offers the same functions as memcached. Additionally to the custom binary-only protocol, we intend to offer authentication and a HTTP based interface to make the service easier to use. There are new API calls for configuration and tailoring the runtime system.

7.2.4.1 Authentication

In order to access any of the functions, an optional authentication step may be necessary, depending on the configuration of the service. Otherwise it isn't possible to execute any of the functions below. Depending on the provided token, access to certain functions or keys may be restricted.

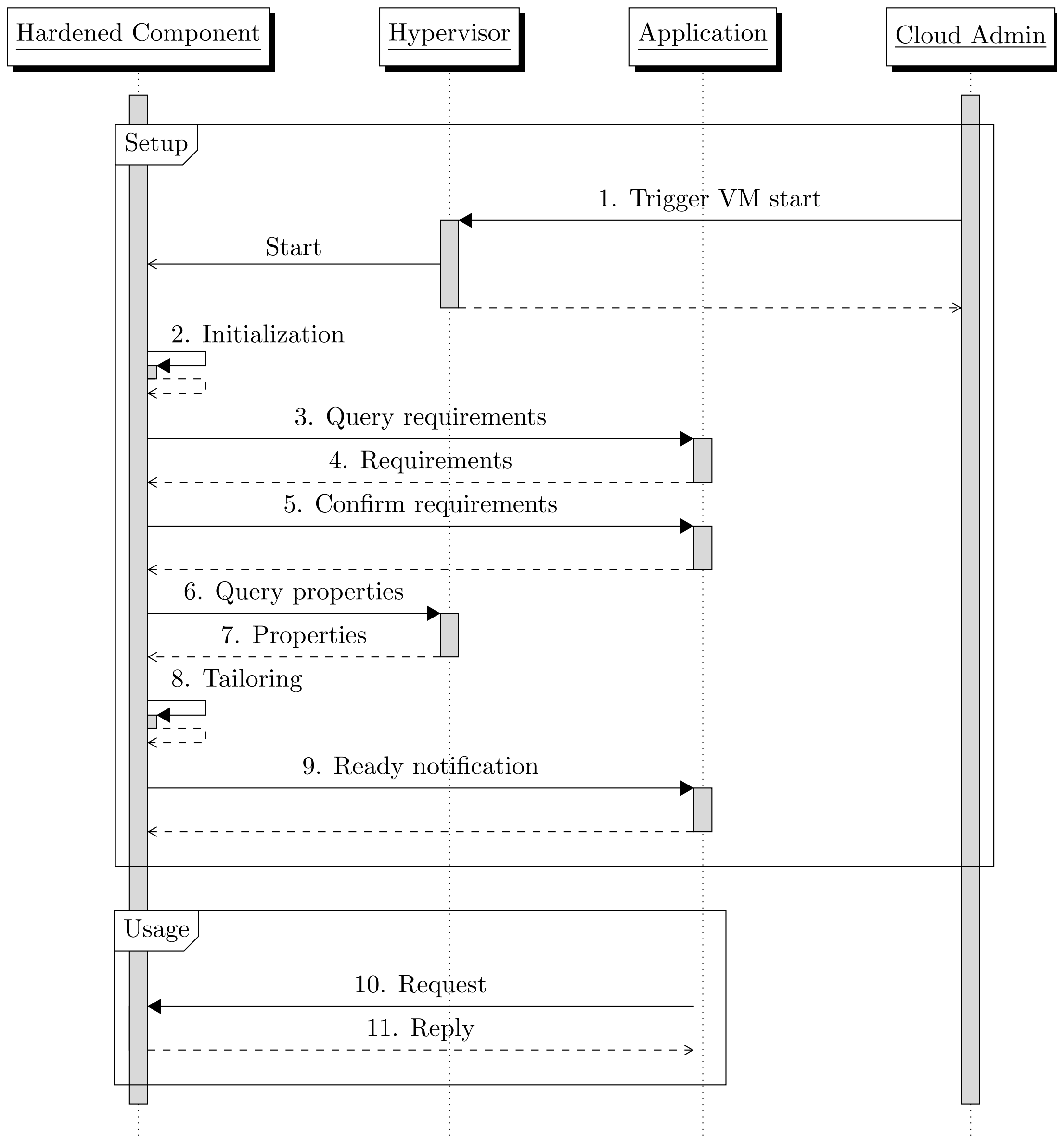


Figure 7.4: Sequence diagram for instantiation of component

```
int authenticate(String token);
```

7.2.4.2 Generic VM image API

In order to tailor the specific service, we provide a interface to set properties of the service instance. This is intentionally completely generic, because the available properties depend entirely on the service application that needs to be tailored.

```
int setProperty(String property, String value);
```

7.2.4.3 key/value store interface

The API proposal for the key/value store sample application closely mirrors the original memcached protocol. It works on CacheEntry objects which are comprised of the following items:

- key: String
- flags: int
- data: byte[]
- expires: int
- casUnique: long

The key entry holds a text string under which the corresponding data array should be saved. The flags field is also stored along with the data and can be used by the application for any purpose. The expires field allows the definition of a timeout in seconds, when the entry will be automatically removed from the cache. When retrieving an entry from the key/value store, it also gets a ”casUnique” tag assigned that is used by the `cas` operation to determine if the entry was changed since its retrieval.

We intend to support at least the following operations:

`ModifyCacheEntry()`

```
int ⇐ set (CacheEntry entry)
```

```
int ⇐ add (CacheEntry entry)
```

```
int ⇐ replace (CacheEntry entry)
```

```
int ⇐ cas (CacheEntry entry)
```

Description.
The functions `set`, `add`, `replace`, and `cas` can be used to set and/or modify values in the key/value store and they return a value that indicates whether the operation was successful. The `set` operation stores the data unconditionally, while `add` will fail if the key already exists. All other storage functions require the key to be already existant. The `cas` function will perform the store operation only if the `casUnique` values of the already stored item and the new entry to be stored also match.

`GetCacheEntries()`

```
CacheEntry ⇐ get (String key)
```

```
CacheEntry[] ⇐ gets (String[] keys)
```

Description.
The `get` functions allow retrieval of the previously stored data and flags value. It also returns the casUnique tag required for the `cas` operation. The `gets` operation allows batched retrieval of multiple keys at once.

`DeleteCacheEntry()`

```
int ⇐ delete (String key)
```

Description.
The `delete` operation immediately removes the specified key from the key/value store.

7.3 Secure Block Storage (SBS)

Authors:
Sven Bugiel, Stefan Nürnberger (TUDA)

7.3.1 Overview

TUDA will contribute *Secure Block Storage* (SBS). Block storage is non-linear raw memory attached to VM instances as block device (virtual hard disk, e.g. iSCSI). SBS will provide a transparent layer that provides security properties such as *confidentiality, integrity* and *authenticity* for block devices. The SBS is also responsible for user-centric key management.

For TClouds two types are relevant:

1. *Public Clouds:* The infrastructure of public clouds cannot be changed. Hence, the security properties must be provided by means established inside the VM. This can for example be achieved by encrypting the blok device, e.g. encryption of Amazon's EBS[2] using TrueCrypt in EC2 instances.

2. *TClouds – OpenStack:* As the infrastructure can be modified, transparent security properties can be added to e.g. the hypervisor in order to provide legacy VM with confidential, integrity-protected and authentic block storage.

We will focus on the latter scenario, because we cannot influence either the storage backend or VM images deployed to a public cloud. The latter solution furthermore has the advantage, that legacy VMs (i.e. VMs not aware of security objectives) can be used, as the modfied hypervisor then functions as a translation layer between ciphertext and plaintext.

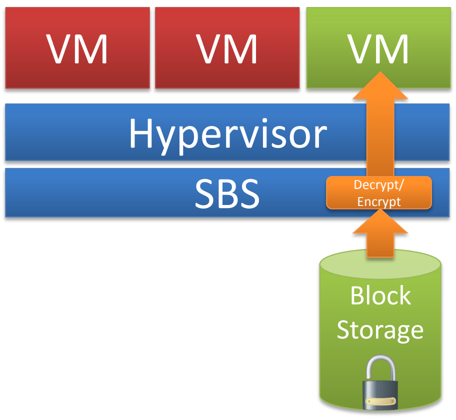


Figure 7.5: Transparent en-/decryption of block storage attached to a VM by the SBS component.

[2]Elastic Block Store

7.3.1.1 Goals (Security, Privacy, Resilience)

- Confidentiality

  Description: The data stored on block devices inside the VM shall be transparently encrypted by the hypervisor so that the stored data at rest cannot be eavesdropped.

  Techniques: Using encryption the stored data (that is mounted as a file system inside the VM) is only accessible in plaintext by those authorized to have access.

  Assumptions: Secure and attestable hypervisor. Otherwise it must be blindly trusted.

- Integrity/Authenticity

  Description: The data stored on block devices inside the VM shall be transparently integrity-protected by the hypervisor so that tampering with the stored data can be detected.

  Techniques: Using digital signatures (or *Message Authentication Codes*, MACs) the stored data[3] can be checked for authenticity and tampering.

  Assumptions: Secure and attestable hypervisor.

- Version Control/Replay Attacks Prevention

  Problem Description: Even though an adversary cannot read encrypted data, it is possible for her to replay previously saved encrypted data. Possible adversaries are: Local/remote administrators of the cloud provider.

  Techniques: Using hardware/virtual counters (e.g. provided by the TPM) it is possible to enable 'version' control for encrypted data chunks.

  Assumptions: Secure and attestable hypervisor.

7.3.1.2 Required External Components

- Trusted Platform.

  Name/description: SBS will rely on a trusted platform with a hardware root of trust. The platform shall provide a hardware Trusted Platform Module (TPM).

  Features (security/privacy/resiliency): Standard TPM features [tcg, tpm].

  Required API (provided by the external component): TPM Interface

- Hypervisor to build on

  Name/description: A Hypervisor that is avaiable in source code, like Xen [xen] or Nova [nov].

  Features (security/privacy/resiliency): Isolation of gues VMs/compartments so that no eavesdropping can occur. As covert channels are currently still subject of research, it unfortunately cannot be assumed that they can be avoided.

  Required API (provided by the external component): Start/Stop VM instances, Trusted Computing enabled, block storage.

[3]to be more precise: a hash thereof

7.3.1.3 Relationship with Activity3

As SBS is transparent, it will not influence Activity3. However, a slightly modified interaction with the actors from A3 is necessary, because they have to additionally provide a cryptographic key.

7.3.2 Requirements

7.3.2.1 Selected Use Cases

USE CASE UNIQUE ID	**/UC 170/ (Provision Encryption Key)**
DESCRIPTION	User securely provisions his encryption key to the SBS component
ACTORS	User
PRECONDITIONS	User has a secret encryption key K
POSTCONDITIONS	Only legitimate Storage Node and User have knowledge of K
NORMAL FLOW	1. User calls *ProvisionKey(k)* function of SBS via interface 2. K together with user-related meta-information are encrypted such that only a legitimate SBS component can decrypt it and provisioned to the cloud infrastructure 3. SBS component of legitimate Storage Node decrypts K and the related meta-information and inserts them into the local management database

USE CASE UNIQUE ID	**/UC 180/ (Create Encrypted Volume)**
DESCRIPTION	User creates a new encrypted volume. This use case extends /UC 40/.
ACTORS	User and Computing Node
PRECONDITIONS	User provisioned his key K (cf. /UC 170/)
POSTCONDITIONS	A new encrypted volume owned by User is created on Storage Node.
NORMAL FLOW	1. A new volume is created (cf. /UC 40/) and flagged with forced de-/encryption using the provisioned key K.

USE CASE UNIQUE ID	**/UC 190/ (Use Encrypted Volume)**
DESCRIPTION	User, inside a VM instance, uses an encrypted volume.
ACTORS	User, Computing Node and Storage Node
PRECONDITIONS	User created the VM instance (cf. /UC 10/) and the volume (cf. /UC 180/). User attached the encrypted volume to the VM instance (cf. /UC 50/). User provisioned his encryption key K (cf. /UC 170/).
POSTCONDITIONS	The VM instance, that has an encrypted volume attached, is able to securely read from and write to the block storage.
NORMAL FLOW	1. VM reads from/writes to the attached block storage 2. SBS component transparently de-/encrypts the data read from/written to the block storage by the VM

7.3.2.2 Demo Storyboard

A user Bob provides a service in the cloud that deals with extremely sensitive data, which is stored persistently on block storage provided by the cloud. To guarantee the confidentiality and integrity of its persistent data, e.g., at rest when the block storage volume is detached, the user leverages the SBS component.

First, the user has to provision his encryption key to the SBS component by invoking the *DeployDataKey* function of management interface (cf. /UC 170/). Hence, his key is now deployed such, that a legitimate SBS component can use it and also identify to which user the key belongs.

Afterwards, the user creates a new secure block storage volume, which is upon access transparently de-/encrypted.

In order to make his VM use the secure block storage volume, he has to attach it to his VM (cf. /UC 190/). All data read from or written to this volume by the user's VM is transparently de-/encrypted with the user's key by the SBS component.

7.3.3 Architecture

7.3.3.1 High-level Design

Is the component required by other A2 components? Yes, by the secure VM image component, which is build on top of this component.

7.3.3.1.1 Hints on implementation

Language mainly C for the hypervisor and Python for OpenStack extension

Existing SW OpenStack; secure Hypervisor for Cloud infrastructure (e.g., Turaya, Nova,...)

7.3.3.2 Sequence Diagrams

Sequence diagram for the key provisioning phase (cf. /UC 170/).

Summary: The *Client* wants to securely provision a key to the SBS component, so that it can be used in by the SBS component only. In order to do so, the SBS component generates a wrapped key, that is, a *"capsule"* which certifies, that a secret can only be used in a pre-defined environment and platform configuration (PCR_{use}). The client can then encrypt (*"encapsulate"*) the secret with the public key of this wrapper. The clients key is then sealed to a known platform configuration for later use.

Predefined functions:

CreateWrapKey(PCR_{use}) Creates a certificate that this wrapped can only be unwrapped in a certain platform configuration (PCR). The wrapping certificate also contains a public key ($.pk$) with wich a secret information can be encrypted. The secret key necessary to decrypt the information again, is only available in the platform configuration (PCR during use – PCR_{use}) specified a priori.

Encrypt(x, pk) Encrypts the message x under asymmetric public key pk

Asymmetric keys (pk, sk) are implied to have already been generated.

7.3.4 API

Public or private? Mainly private, since mainly transparent. However, a minimal API is public to enable clients to deploy encrypted data and supply the corresponding key securely to the infrastructure.
New or extended? The interface to the client is an extension to the OpenStack API.

7.3.4.1 SBS API

`PrepareProvisioning()`

```
ProvisionStruct ⇐ PrepareProvisioning ()
```

Description.
The function `PrepareProvisioning` sets up a trusted channel between the Client and the SBS component. It requests a *Certified Binding Key* in order for the Client to be able to securely deploy his secret key to the SBS component. The function returns an object of type `ProvisionStruct`.

The struct `ProvisionStruct` is composed of:

bindKey A TPM WrapKey structure that holds

- **Public key:** In order for the client to encrypt his key (k in Figure 7.6).
- **Encrypted Secret Key:** Only available to the TPM to decrypt encrypted data supplied by the Client.
- **Meta Information:** Platform state at time of creation and allowed usage

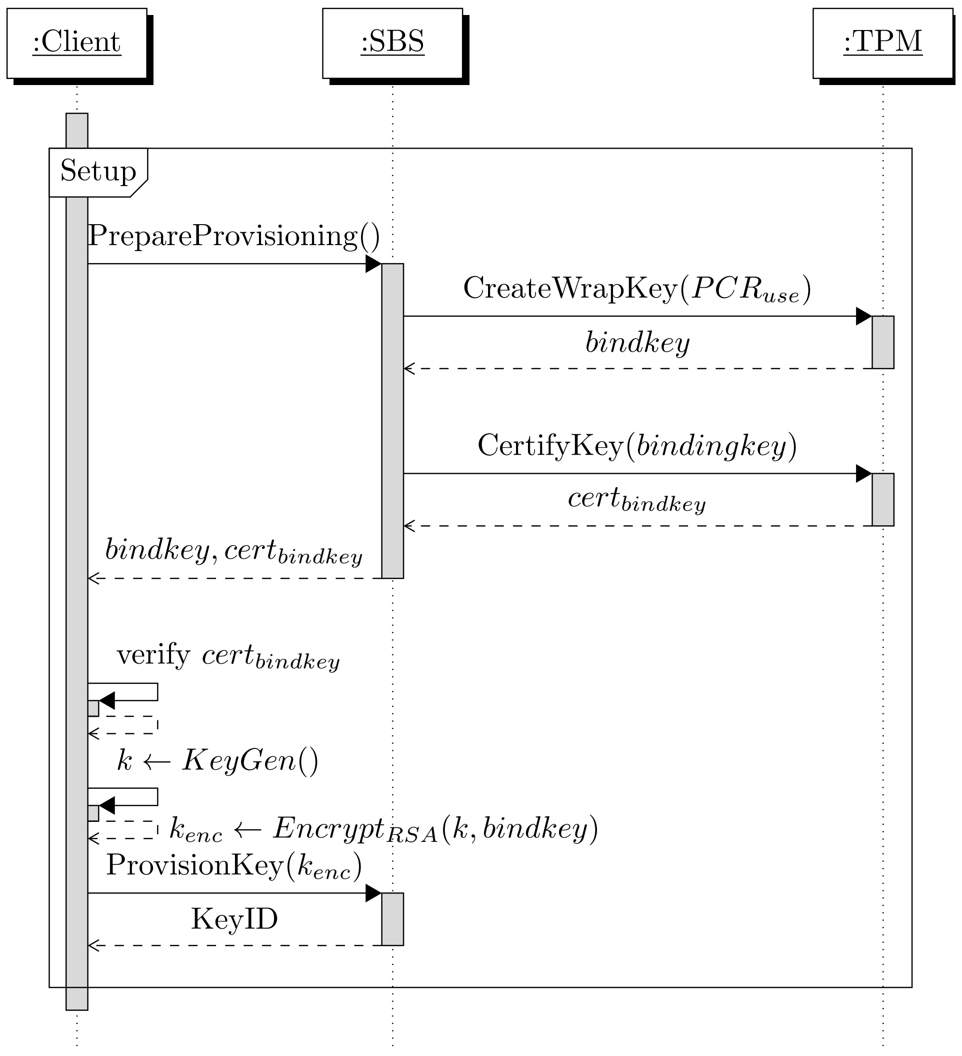


Figure 7.6: Sequence Diagram of the Setup Phase (key provisioning) for the SBS component

certificate A certificate created by the TPM that certifies the *bindingkey* was created by a legitimate TPM chip.

`ProvisionKey()`

```
void ⇐ ProvisionKey (KeyStruct key)
```

Description.
After the client has verified the legitimate origin of the bind key (using the certificat) it can confirm the usage of the bindingkey using the Meta Information. The client can then encrypt its key using the *bindingkey* and deploy it using the `ProvisionKey` function.

Functions to actually deploy data to block storage are inherited from the OpenStack API.

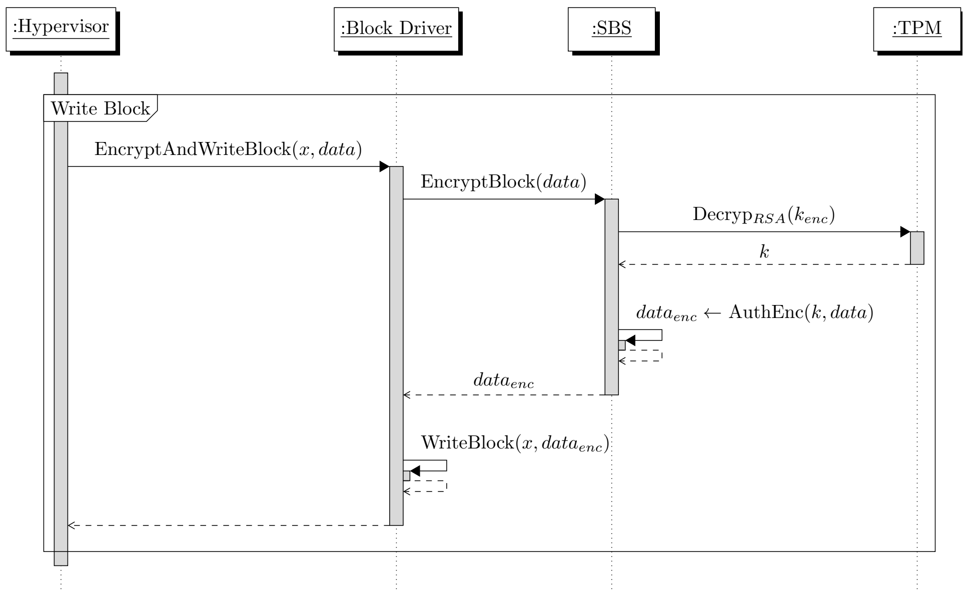

:Hypervisor
:Block Driver
:SBS
:TPM
Write Block
EncryptAndWriteBlock(x, $data$)
EncryptBlock($data$)
Decryp$_{RSA}$(k_{enc})
k
$data_{enc} \leftarrow$ AuthEnc(k, $data$)
$data_{enc}$
WriteBlock(x, $data_{enc}$)

Figure 7.7: Sequence Diagram of SBS component writing encrypted data

7.4 Secure VM Instances

Authors:
Sven Bugiel, Stefan Nürnberger (TUDA)

7.4.1 Overview

Based on the secure block storage component (SBS), TUDA will contribute with a component that allows clients to securely deploy, launch, and migrate their own VM images. The component ensures that the VM images and data contained within will be confidentiality and integrity protected when they are at rest in a image repository or in transit during migration. The authenticity can be ensured using a secure channel.

7.4.1.1 Goals (Security, Privacy, Resilience)

- Confidentiality

  Description VM images, especially data contained within, must be protected against eavesdropping, e.g., by a remote administrators at the cloud service provider. The key used to encrypt the images at client-side is bound to a trusted hypervisor configuration.

  Techniques/research problems The secure block storage component will be used and extended with corresponding interfaces and functionality.

  Assumptions Secure and attestable hypervisor (as for SBS).

- Integrity

  Description Modifications to the VM images at rest must be detected.

  Techniques/research problems The secure block storage component will be used and extended with corresponding interfaces and functionality.

Assumptions Secure and attestable hypervisor (as for SBS).

7.4.1.2 Required External Components

- Component1

  Name/description Secuer block storage (SBS)

  Features (security/privacy/resiliency) Confidential, integrity protected, and authenticated block storage

  Required API (provided by the external component) API to store and retrieve VM images from an image repository

7.4.1.3 Relationship with Activity3

Actors from A3 can securely deploy their VM images to an image repository in the cloud and deploy the necessary encryption in the cloud infrastructure.

7.4.2 Requirements

We assume that

1. The cloud administrator does not eavesdrop the VM memory, as it may contain plaintext
2. A trusted path to the client in order to securely deploy the key

7.4.2.1 Selected Usecases

USE CASE UNIQUE ID	**/UC 200/ (Create Secure Image)**
DESCRIPTION	User deploys his VM image securely in the cloud. This use case extends /UC 30/.
ACTORS	User
PRECONDITIONS	None.
POSTCONDITIONS	User's VM image securely deployed in the cloud infrastructure
NORMAL FLOW	1. User encrypts his VM image with key K 2. User deploys the encrypted image by calling the interface function `RegisterImage(Data, KeyID)`.

USE CASE UNIQUE ID	**/UC 210/ (Create Instance)**
DESCRIPTION	User starts his VM image. This use case extends /UC 10/.
ACTORS	User, SBS component, hypervisor
PRECONDITIONS	User has provisioned his key K to the SBS component (cf. /UC 170/) and deployed his encrypted image (cf. /UC 200/).
POSTCONDITIONS	User's VM image is instantiated as VM instance
NORMAL FLOW	1. User issues the VM start command to the cloud management interface 2. The hypervisor starts the user's VM image 3. The SBS component de-/encrypts the VM image during execution

USE CASE UNIQUE ID	**/UC 220/ (Migrate Instance)**
DESCRIPTION	The user's VM instance is migrated to a new physical host in the cloud infrastructure. This use case extends /UC 100/.
ACTORS	Hypervisor, SBS component, (optional: User)
PRECONDITIONS	User has provisioned his key K to the SBS component
POSTCONDITIONS	User's VM is migrated to a new host
NORMAL FLOW (AUTOMATIC MIGRATION)	1. The hypervisor currently executing the user's VM instance initiates the migration of the instance to a new host 2. The SBS component on the original host migrates the user's encryption key K to the new host (if the key is not yet available there). 3. The VM is migrated and executes on the new host, whose hypervisor has access to the VM instance via its SBS component and the previously migrated key K.
ALTERNATIVE FLOW (USER INITIATED MIGRATION)	1. The user initiates the migration of his VM 2. Identical to the normal flow

7.4.2.2 Demo Storyboard

To extend the storyboard from 7.3.2.2, the user additionally wants to protect sensitive information within his VM when this VM is "at rest", i.e., the VM image. Those credentials are protected at run-time by, e.g., a root-less and trusted hypervisor, but are at rest prone to tampering.

The user re-uses his previously deployed encryption key (cf. /UC 170/). However, the user is first required to provision his encrypted VM image, that shall run in the cloud (cf. /UC 200/).

The image is encrypted with the previously deployed user's key. Afterwards, the user can instantiate his VM image (cf. /UC 210/) as a new instance. The SBS component transparently de-/encrypts the image during execution with the user's key.

If the VM instance has to be migrated to a new host in the cloud, e.g., due to load-balancing, the new host requires the user's key in order to access the VM instance. The key is thus migrated previously to the VM instance (cf. /UC 220/).

7.4.3 Architecture

7.4.3.1 High-level Design

Is the component required by other A2 components? Probably not.

7.4.3.1.1 Hints on implementation

Language Mostly C for the extension of SBS, Python for the extension of OpenStack with the new public API

Existing SW same as for SBS

7.4.3.2 Sequence Diagrams

Sequence diagram for VM image access/booting (cf. /UC 190/).

Summary: The *Client* wants to start a VM that belongs to him/her. Therefore, the Hypervisor issues a VM Boot Phase. The corresponding block device driver is re-routed to decrypt the raw blocks before they are sent to the hypervisor in order to access the image/boot the image. The Client's key k that is used inside the SBS is only available if the SBS's code base matches the one the client has verified (see key wrapping and PCR_{use} in Sequence Diagram depicted in Figure 7.6). After the integrity, authenticity and freshness has been verify $(DecVerify)$ the message can be decrypted and is relayed to the Hypervisor.

Predefined functions:

DECRYPTKEY(k_{enc}) Can decrypt a wrapped key (k_{enc}) when the platform configuration of its use (PCR_{use}, see subsubsection 7.3.3.2) is met.

DECRYPT(sk, x) Decrypts the ciphertext x given the secret key sk.

DECVERIFY(k, x) Decrypts the ciphertext x using the symmetric key k, if and only if the verification of the embedded MAC succeeded. That means, that the authenticity and integrity could be verified. DECVERIFY is the opposite of AUTHENC, the authenticated encryption.

Asymmetric keys (pk, sk) are implied to have already been generated.

7.4.4 API

Public or private? The API to the necessary extensions to SBS is public.
New or extended? Extension of SBS
Public or private? Mainly private, since mainly transparent. However, a minimal API is public to enable clients to deploy encrypted data and supply the corresponding key securely to the infrastructure.
New or extended? The interface to the client is an extension to the OpenStack API.

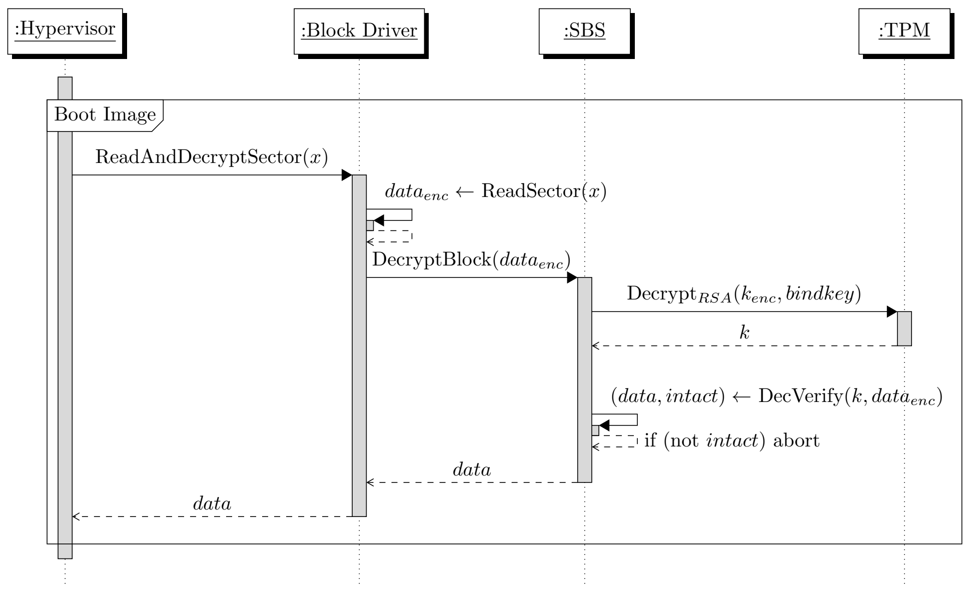

:Hypervisor
:Block Driver
:SBS
:TPM
Boot Image
ReadAndDecryptSector(x)
$data_{enc}$ ← ReadSector(x)
DecryptBlock($data_{enc}$)
Decrypt$_{RSA}$(k_{enc}, $bindkey$)
k
($data$, $intact$) ← DecVerify(k, $data_{enc}$)
if (not $intact$) abort
$data$
$data$

Figure 7.8: Sequence Diagram of the Boot Phase (Hypervisor starts image)

7.4.4.1 VM Image API

RegisterImage()

```
ImageID ⇐ RegisterImage (Data, KeyID)
```

Description.
In general the APIs for VM Image management (e.g. register images, instantiate images, delete images) are inherited from the OpenStack API. However, the registration of a new image requres the client to encrypt the image before deployment and to associate it with the corresponding and already provisioned key. The registration returns an `ImageID` identifying the image, which can later be used to refer to that image.

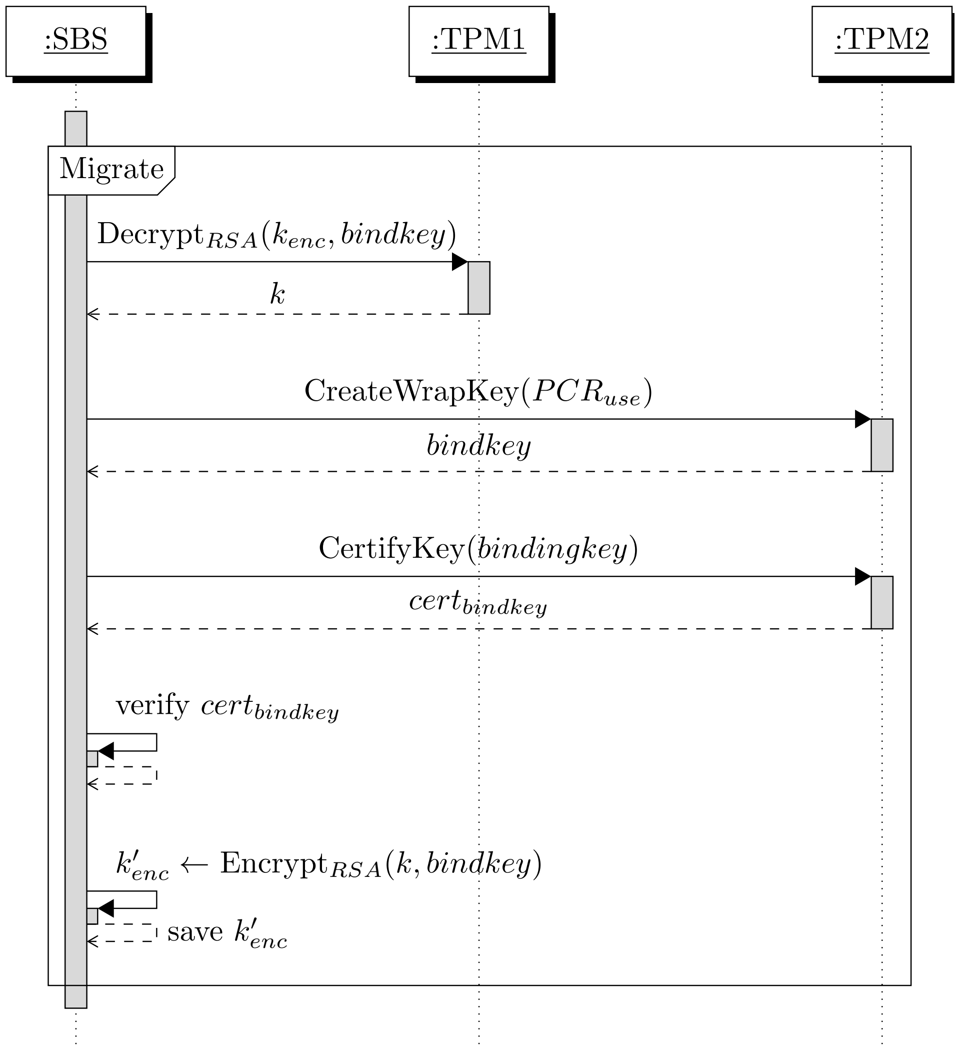

:SBS
:TPM1
:TPM2
Migrate
$\text{Decrypt}_{RSA}(k_{enc}, bindkey)$
k
$\text{CreateWrapKey}(PCR_{use})$
$bindkey$
$\text{CertifyKey}(bindingkey)$
$cert_{bindkey}$
verify $cert_{bindkey}$
$k'_{enc} \leftarrow \text{Encrypt}_{RSA}(k, bindkey)$
save k'_{enc}

Figure 7.9: Sequence Diagram of the key migration

7.5 TrustedServer

Authors:
Michael Gröne, Norbert Schirmer (SRX)

7.5.1 Overview

This section provides an overview of the TrustedServer (TS) component. In the following we describe the overall goals and requirements, and provide an analysis of its security.

7.5.1.1 Description

SRX will provide the TrustedServer as the central security platform to run the VM instances (also called compartments). It is based on the TURAYA^{TM} SecurityKernel and provides isolation of compartments by linking them to TVD s. Domain specific transparent encryption is applied to prohibit information flow between TVDs. The focus of this component is to provide (together with TrustedObjects Manger (TOM; cf. 9.2)) a trusted platform for cloud applications from the ground up.

7.5.1.2 Goals (Security, Privacy, Resilience)

- TVD enforcement for VM instances
- Integrity: remote attestation of server configuration via TPM.
- Confidentiality: transparent encryption of data to prohibit undesired information flow between TVDs.
- Restricted administrator rights: no almighty root account on server. Server is managed remotely via TOM (cf. 9.2). Security Services provided by server only have the necessary rights needed for their task. **Description:** With restricted administrator rights combined with the Component 'Secure VM Instances' (cf. 7.4), we aim at a 'black-box' view of VM instances for the cloud provider.
- Optional: Integration with OpenStack **Description:** The main goal is to provide a replacement for OpenStack. However, we have to evaluate if and how OpenStack components can be integrated / extended. Simply putting OpenStack on top of a TrustedServer is definitely not enough. Currently we use VirtualBox as virtualization layer, which is not supported by OpenStack right now. Moreover the management layers (TOM and OpenStack) have to be integrated.

7.5.1.3 Required External Components

The platform the component is installed on shall provide a hardware Trusted Platform Module (TPM), which could be replaced by a Hardware Security Module (HSM) in the future.

7.5.1.4 Relationship with Activity3

A TrustedServer is a core infrastructure element of a trusted cloud, and hence used by any application. In year 3 we should be capable to run the applications on this platform (as alternative to OpenStack).

7.5.2 Requirements

This section gives an overview of the requirements for the TrustedServer component.

7.5.2.1 Preconditions

Requirements that have to be fulfilled already, because they were needed for the development process.

/PR 10/ Trusted bootloader
A bootloader with TPM-support is required.

/PR 20/ TIS-Driver
A TIS-Driver is required to use TPMs of version 1.2.

7.5.2.2 Execution Environment

This section specifies software and hardware the user requires at least to run the component successfully.

7.5.2.2.1 Hardware

- TPM 1.2 Platform

7.5.2.2.2 Software

- TrustedServer

7.5.2.2.3 Infrastructure

- TrustedObjects Manager
- Trusted Management Channel

7.5.2.3 Security Environment

This section describes the security aspects of the environment in which the component is intended to be used and the manner in which it is expected to be employed.

7.5.2.3.1 Assumptions A description of assumptions shall describe the security aspects of the environment in which the component will be used or is intended to be used.

/A 10/ Trusted Organization Administrator
The organization administrator of the managed IT infrastructure is non-malicious.

/A 20/ Trusted Administrator
The security administrator of the system is non-malicious.

/A 30/ Correct hardware
The underlying hardware (e.g., CPU, devices, TPM) does not contain backdoors, is non-malicious, and behaves as specified.

/A 40/ Attestation

The IT-environment provides a mechanism that allows the component to convince remote parties about its trustworthiness. Example mechanisms are to perform an attestation protocol based on an environment providing authenticated boot. Another example would be a tamper-resistant hardware environment that can uniquely by identified as such be a remote party, e.g., based on a signature key stored inside.

/A 50/ TOE Binding

The IT-environment offers a mechanism that allows the component to store information such that it cannot be accessed by another component configuration. Example mechanisms are the sealing function offered by a TPM as specified by the TCG in combination with an authenticated bootstrap architecture, or a tamper-resistant storage in combination with a secure bootstrap architecture.

/A 60/ No man-in-the-middle attack

A physical attack that relays the whole communication between a local user and the I/O devices to another device does not happen.

/A 70/ Untrusted Cloud Administrator

The Cloud Admin of the system may be malicious.

7.5.2.3.2 Assets This section defines the sensitive information the security kernel is operating on.

/AS 10/ Identity Key

The identity key is created during the production process and used during the whole lifetime to identify the component.

/AS 20/ Encryption Key

The encryption key is created during the production process and used, e.g., to decrypt firmware updates.

/AS 30/ Trusted Virtual Domain Keys

The Trusted Virtual Domain key is created during the instantiation of a compartment and is used, e.g. for the transparent file encryption.

7.5.2.4 Security Objectives

The security objectives address all of the security environment aspects identified. The security objectives reflect the stated intent and shall be suitable to counter all identified threats and cover all identified organizational security policies and assumptions. A threat may be countered by one or more objectives for the component, one or more objectives for the environment, or a combination of these.

7.5.2.4.1 Security Objectives of the IT-Environment

/OE 10/ TrustedServer Integrity Prove

The IT-environment provides a mechanism that allows the TrustedServer to convince remote parties and local users about its integrity. Common examples of such a mechanism are a secure bootstrap architecture as used, e.g., by the AEGIS architecture [SCG+03], or an authenticated bootstrap architecture as specified by the TCG [tpm].

/OE 20/ Backup

The IT-environment ensures that the information stored by the TrustedServer is backuped in regular intervals.

7.5.2.4.2 Security Objectives for the TrustedServer

/O 10/ TrustedServer Identity

Using functionalities offered by the IT-Environment, the TrustedServer should be able to prove its identity to both remote parties and local users.

/O 20/ TrustedServer Integrity

Using the functionalities offered by the IT-Environment, the TrustedServer should be able to convince remote parties and local users that the integrity of the TrustedServer is not violated.

Changes in the TrustedServer must be detectable by both the user and remote parties. Such changes can drastically affect the security properties of the system, and therefore mechanisms must be put in place to prevent entrusting sensitive data to such a compromised system.

/O 30/ Strong Isolation

The TrustedServer should strongly isolate compartments from each other. The isolation has to be enforced on the address-space level and on the data level. More concretely: the use of different compartments has to be at least as secure as the execution of the same applications on physically separated computing platforms connected via network.

/O 40/ Admin Authentication

The TrustedServer should always identify and authenticate administrators before granting access to management functions of the TrustedServer.

/O 50/ User Authentication

Depending on the underlying security policy, the TrustedServer should be able to identify and authenticate users before granting access to compartments.

/O 60/ Trusted Channel Between Compartments

The TrustedServer should provide a trusted communication channel between compartments, i.e., a channel providing integrity, confidentiality, and authenticity of the compartment's configuration.

/O 70/ Trusted Path to Users

The TrustedServer should provide a trusted communication channel, i.e., a channel providing integrity, confidentiality, and authenticity of the compartment's configuration, between compartments and local users. Moreover, the TrustedServer should provide a trusted communication channel between itself and local users.

/O 80/ Secure Persistent Storage

The TrustedServer should provide data containers to persistently store information providing (at least) the following list of security properties:

- *Integrity:* Allow the compartment to detect an integrity violation.
- *Confidentiality:*
  - *TrustedServer:* Allow a compartment to bind information to the TrustedServer.
  - *Compartment:* Allow a compartment to bind information to a compartment configuration.
  - *Role:* Allow a compartment to bind information to a specific user role.

  Freshness: Allow compartments to store information such that a replay attack can be detected.

/O 90/ Data Availability after TrustedServer Update

The TrustedServer should ensure the availability of user data not bound to a specific TrustedServer version after a TrustedServer update providing the same security properties.

/O 100/ Data Availability after Compartment Update

The TrustedServer should ensure the availability of user data not bound to a specific compartment version after a compartment update providing the same security properties.

/O 110/ Data Availability after TrustedServer migration

The TrustedServer should ensure the availability of user data not bound to a specific TrustedServer version after a migration to another TrustedServer providing the same security properties.

/O 120/ Data Availability after IT-environment migration

The TrustedServer should ensure the availability of user data not bound to a specific IT- Environment after a migration to another IT-Environment providing the same security properties.

- an update of the TrustedServer,
- an update of a Compartment,
- a migration to another TrustedServer,
- a migration to another IT-environment

7.5.2.5 Security Requirements

This part defines the security requirements that have to be satisfied by the component. The statements shall define the functional and assurance security requirements that the component and the supporting evidence for its evaluation need to satisfy in order to meet the security objectives.

/SR 10/ Integrity of the TCB
The TCB should be protected from manipulations to guarantee the enforcement of security policies. No modification of the TCB must be allowed, except for changes that have been authorized by the Admin.

/SR 20/ Confidentiality and Integrity of Application Data
Application data should remain confidential and integer during execution and storage.

/SR 30/ Trusted Path to User
The inputs/outputs of the application a user interacts with should be protected from unauthorized access by other applications.

/SR 40/ Trusted Channel between Trusted Compartment and External Parties
Trusted channels must be provided to allow remote parties to interact with the TrustedServer system while being assured of its well-behavior and its willingness to conform to their security policy.

/SR 50/ Information Flow
Information flow should only be possible where allowed by the security policy[4]. Primarily, evesdropping on another, non-cooperating compartment must be foiled.

7.5.2.6 Selected Usecases

[4]Covered channels may still exist, but due diligence must be taken to minimize their impact.

USE CASE UNIQUE ID	**/UC 230/ ((Semi-)Initial Connect To TOM)**
DESCRIPTION	The TrustedServer is booted and connects to TOM
ACTORS	Cloud Admin, TrustedChannel, TOM
PRECONDITIONS	TrustedServer is installed and registered at TOM, TOM is configured for managing this TrustedServer, TOM is running
POSTCONDITIONS	TrustedServer is connected to TOM via a TrustedChannel
NORMAL FLOW	1. Cloud Admin boots a TrustedServer 2. TrustedServer is booted 3. After booting TrustedServer tries to connect to TOM 4. TrustedServer initiates connection to TOM 5. Remote Attestation succeeds 6. Connection is established (via TrustedChannel)

7.5.2.7 Demo Storyboard

Here we demonstrate how a Cloud Admin initially connects a TrustedServer to TOM:

1. Cloud Admin opens TOM's Management Interface within a web-browser
2. Status of pre-registered TrustedServer is 'offline'
3. Cloud admin starts the TrustedServer
4. The TOM-log shows the (plaintext) objects, transmitted via the TrustedChannel
5. After a successful Remote Attestation, the predefined configuration for the TrustedServer sent by TOM can be seen in the log-file of the TrustedServer
6. The log of TrustedServer shows the successful application of the configuration
7. TOM's Management interface shows an 'online' status of the TrustedServer

7.5.3 Architecture

In this section the high-level design and sequence diagrams of the TrustedServer component are described.

7.5.3.1 High-level Design

- Big picture and relations with other components
  - The component is managed by TOM (WP 2.3).
  - Integration of 'Secure VM Instances' component is desired.
- Hints on implementation
  - none

7.5.3.2 Sequence Diagrams

The following sequence diagrams describe the interactions between the (sub)components involved when using the TrustedServer component. They implement the use cases /UC 230/ defined in section 7.5.2.6 and the common use case /UC 20/ . The functions used in the sequence diagrams comes from the API discussed in section 7.5.4.

7.5.3.2.1 Setup (Figure 7.10). To setup TrustedServer the Cloud Admin will download a configuration from the TOM and then start the TrustedServer (cf. Figure 7.10) to bind it to TOM.

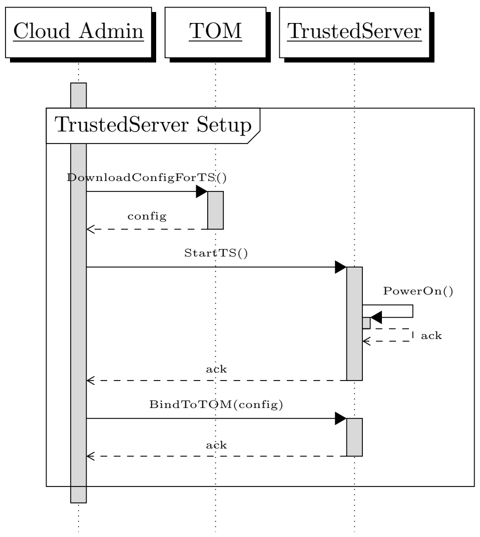


Figure 7.10: Setup of TrustedServer.

7.5.3.2.2 Start Compartment (Figure 7.11). To start a compartment on a TrustedServer the TOM sends a start command to TURAYA^{TM} Manager which is part of secure hypervisor and will boot the VM instance (cf. Figure 7.11). This is an instance of Usecase /UC **??**/. k

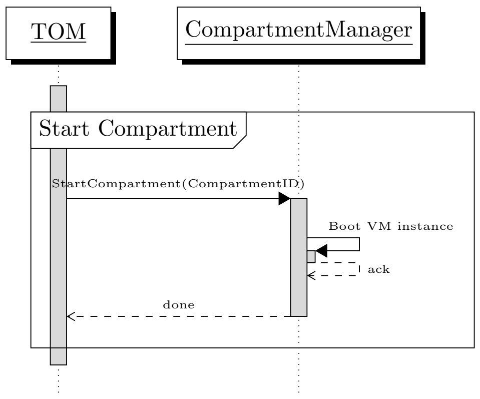


Figure 7.11: Start of compartment on TrustedServer.

7.5.4 API

In this section we describe high-level API for the TrustedServer compartment management functions 7.12).

- Public interface via TrustedChannel for remote connection(s) to TOM(s).
- (Binary) Management protocol (within TrustedChannel) for management tasks.

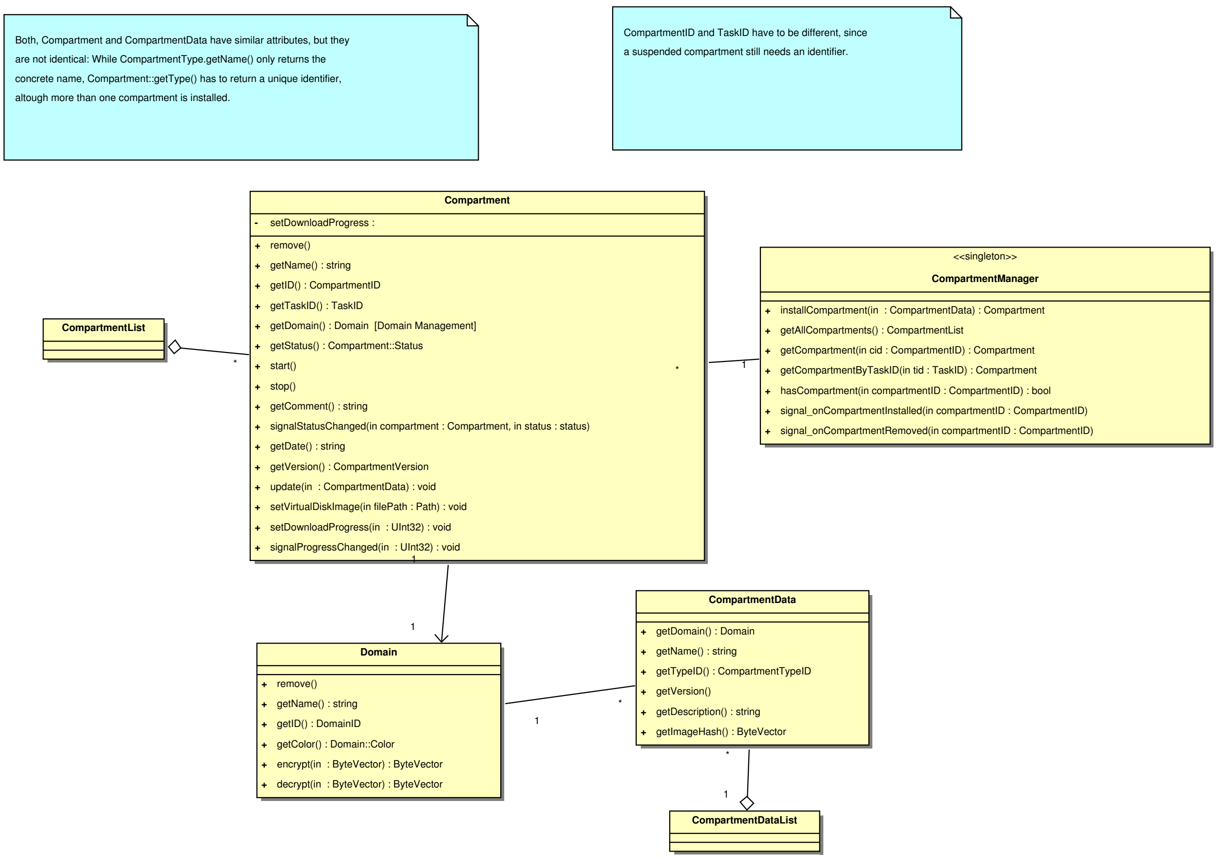

Both, Compartment and CompartmentData have similar attributes, but they are not identical: While CompartmentType.getName() only returns the concrete name, Compartment::getType() has to return a unique identifier, altough more than one compartment is installed.
CompartmentID and TaskID have to be different, since a suspended compartment still needs an identifier.
Compartment
- setDownloadProgress :
+ remove()
+ getName() : string
+ getID() : CompartmentID
+ getTaskID() : TaskID
+ getDomain() : Domain [Domain Management]
+ getStatus() : Compartment::Status
+ start()
+ stop()
+ getComment() : string
+ signalStatusChanged(in compartment : Compartment, in status : status)
+ getDate() : string
+ getVersion() : CompartmentVersion
+ update(in : CompartmentData) : void
+ setVirtualDiskImage(in filePath : Path) : void
+ setDownloadProgress(in : UInt32) : void
+ signalProgressChanged(in : UInt32) : void
CompartmentList
<<singleton>>
CompartmentManager
+ installCompartment(in : CompartmentData) : Compartment
+ getAllCompartments() : CompartmentList
+ getCompartment(in cid : CompartmentID) : Compartment
+ getCompartmentByTaskID(in tid : TaskID) : Compartment
+ hasCompartment(in compartmentID : CompartmentID) : bool
+ signal_onCompartmentInstalled(in compartmentID : CompartmentID)
+ signal_onCompartmentRemoved(in compartmentID : CompartmentID)
Domain
+ remove()
+ getName() : string
+ getID() : DomainID
+ getColor() : Domain::Color
+ encrypt(in : ByteVector) : ByteVector
+ decrypt(in : ByteVector) : ByteVector
CompartmentData
+ getDomain() : Domain
+ getName() : string
+ getTypeID() : CompartmentTypeID
+ getVersion()
+ getDescription() : string
+ getImageHash() : ByteVector
CompartmentDataList
*
1
*
1
1
1
1
*
*
1

Figure 7.12: Compartment Management API.

7.5.4.1 CompartmentManager

This section comprised CompartmentManager sub-component interface for Install-Compartment / Remove-Compartment / Start-Compartment / Stop-Compartment.

install()

```
installResult ⇐ install (CompartmentImage, CompartmentConfig)
```

Description.
The `install` API is called by the Cloud Admin to install a Compartment on TrustedServer.

remove()

```
removeResult ⇐ remove (CompartmentID)
```

Description.
The `remove` API is called by the Cloud Admin to delete an installed Compartment-instance on TrustedServer.

start()

```
startResult ⇐ start (CompartmentID)
```

Description.
The `start` API is called by the Cloud Admin to start an installed Compartment-instance on TrustedServer.

stop()

```
stopResult ⇐ stop (CompartmentID)
```

Description.
The `stop` API is called by the Cloud Admin to stop an Compartment-instance on TrustedServer.

7.5.4.2 API Parameters and Return Values

Parameter	Description
`CompartmentID`	An unique identifier for this compartment (int)
`CompartmentImage`	Path/to/filename of VM-image-file (vdi)
`CompartmentConfig`	The configuration to be applied to the CompartmentImage-instance: Name:string ; Description:string ; ImageHash:ByteVector ; Domain:DomainObject ; TypeID:CompartmentTypeID

Return Values	Description
`installResult`	Represents the result related to the performed installation.
`removeResult`	Represents the result related to the performed deletion.
`startResult`	Represents the result related to the performed start operation.
`stopResult`	Represents the result related to the performed stop operation.

Verification and Auditability

7.6 Log Service

Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

7.6.1 Overview

Log Service is the TClouds logging subsystem, mainly used by other Cloud Components to log their internal events and, possibly, by applications. Log Service can be used as basis for auditing or reporting the Service Level Agreement (SLA) compliance to the User (here the main target of the service is the end user of the cloud, but it may also refer to an external auditor or to the Cloud Admin). In WP2.1, we concentrate in providing integrity and privacy of logs through access control mechanisms, whilst in WP2.2, we concentrate on ensuring their availability and logging of cloud of clouds events.

7.6.1.1 Goals (Security, Privacy, Resilience)

- Integrity of log entries.
  Description: Log Service will protect log entries with strong cryptographic methods so that the User will immediately detect tampering of log entries.
  Techniques/research problems: Log Service will employ techniques such as those proposed by Schneier and Kelsey [SK99] or Ma and Tsudik [MT09].
  Assumptions: Log Service does not need a secure storage for protecting log entries.

- Privacy and access control of log entries.
  Description: Log Service will create log entries with privacy enforced by design, i.e. with all the sensitive information already removed or protected. Moreover, mechanisms for ensuring access control on log entries will be used.
  Techniques/research problems: Log Service will employ techniques such as k-anonymity or broadcast encryption.
  Assumptions: Log Service does not need a secure storage for protecting log entries.

- Availability of logs.
  Description: Log Service will be capable of guaranteeing availability of logs, also for long periods of time.
  Techniques/research problems: still to be defined.
  Assumptions: Log Service requires a resilient storage.

- Policies on log entries.
  Description: Log Service will be capable of applying policies on log entries that define their usage. For instance, a policy may specify that personal data must be kept at most 6 months, while medical records must be kept at least 10 years.
  Techniques/research problems: still to be defined.
  Assumptions: Log Service requires a resilient storage.

7.6.1.2 Required external components

- Resilient storage (cf. Section 8.3).
  Name/description: the resilient storage is needed because log entries are supposed to be available and to last for long time. To the opposite, Log Service will protect the integrity and the confidentiality of log entries by itself without relying on a secure storage for such a task.
  Features (security/privacy/resiliency): the storage must be resilient.
  Required API (provided by the external component): Log Service does not require any specific API and will use the one defined for storage.

- Trusted platform.
  Name/description: Log Service will rely on a trusted platform with a hardware root of trust.
  Features (security/privacy/resiliency): the trusted platform must provide isolation of critical components of Log Service from the rest of the (untrusted) system. Moreover, it must be equipped with a hardware root of trust capable of securely store and use cryptographic keys (a TPM should suffice). Moreover, if the platform is capable or reporting its integrity is a plus.
  Required API (provided by the external component): Log Service will be based on TSS, but we will probably also use the TPA [CCS$^+$11, see also D2.1.1, Chapter 13].

7.6.1.3 Relationship with Activity3

Actors from A3 act as end users of the cloud and therefore can access the Log Service. For instance, an A3 Project Manager may want to check if his application deployment complies with the SLA, using the Management Console provided by TClouds.

The current API is only intended to be used by Cloud Components (i.e., by A2 components), but this can be extended to be used by applications as well.

7.6.2 Requirements

The use cases are depicted in Figure 7.13. We define the following terminology:

- *log entry*: a record containing information about an event. The log entry may give only a partial view of the event. Moreover (part of) the data may be sensitive.

- *event log*: a (usually small) set of log entries all related to a single event. This is the smallest set that provides the overall view on the event.

- *log (or registry log)*: a set of log entries, usually all related to a single object or actor. Note that a single log entry may be added to several log registries.

7.6.2.1 Selected Use Cases

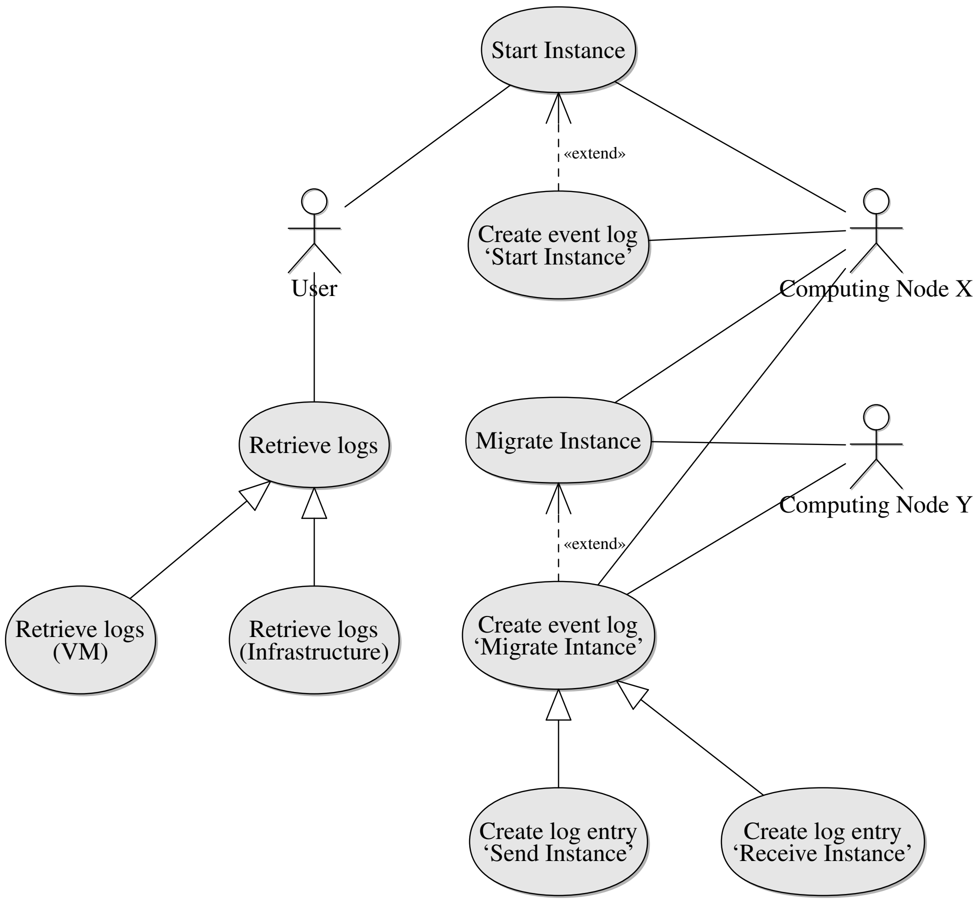

Start Instance
«extend»
Create event log 'Start Instance'
User
Computing Node X
Retrieve logs
Migrate Instance
Computing Node Y
«extend»
Retrieve logs (VM)
Retrieve logs (Infrastructure)
Create event log 'Migrate Intance'
Create log entry 'Send Instance'
Create log entry 'Receive Instance'

Figure 7.13: Use case diagram to demonstrate the Log Service at cloud infrastructure level.

USE CASE UNIQUE ID	**/UC 240/ (Create event log 'Start Instance')**
DESCRIPTION	A new event log is created to track the event of starting a VM instance.
ACTORS	Computing Node X.
PRECONDITIONS	User started a VM instance (e.g., VM$_{A1}$) on Computing Node X, cf. /UC 20/.
POSTCONDITIONS	None.
NORMAL FLOW	1. Computing Node X creates a new log entry for the VM instance being started containing "started instance VM$_{A1}$". 2. Computing Node X adds the log entry to its log. 3. Computing Node X adds the log entry to the log of VM$_{A1}$.

USE CASE UNIQUE ID	**/UC 250/ (Create event log 'Migrate Instance')**
DESCRIPTION	A new event log is created to track the event of migrating a VM instance.
ACTORS	Computing Node X and Computing Node Y.
PRECONDITIONS	Migration of VM_{A1} from Computing Node X to Computing Node Y has been triggered, cf. /UC 100/
POSTCONDITIONS	None.
NORMAL FLOW	1. Before sending VM_{A1}, Computing Node X: (a) creates a new log entry for the VM instance migration containing "Sent VM_{A1} from Computing Node X to Computing Node Y". (b) adds the log entry to its log. (c) adds the log entry to the log of VM_{A1}. 2. Upon successfully receiving VM_{A1}, Computing Node Y: (a) creates a new log entry for the VM instance migration containing "Received VM_{A1} from Computing Node X to Computing Node Y". (b) adds the log entry to its log. (c) adds the log entry to the log of VM_{A1}.

USE CASE UNIQUE ID	**/UC 260/ (Retrieve logs)**
DESCRIPTION	A log is retrieved by the User.
ACTORS	User.
PRECONDITIONS	None.
POSTCONDITIONS	None.
NORMAL FLOW (LOG OF A VM)	1. The log of the requested VM instance is provided to the User together with data necessary to authenticate the log (e.g. cryptographic keys).
ALTERNATIVE FLOW (LOG OF THE INFRASTRUCTURE)	1. Log Service collects logs from the portion of infrastructure that pertains to User (e.g. Computing Nodes where his VM instances have been executed). Note that parts of these logs may be sensitive for the Cloud (e.g. detailed information about the physical node) or for other users that have been using the infrastructure (e.g. information about their VM instances). In this case, the confidentiality of sensitive data must be guaranteed. 2. The logs of the infrastructure are provided to User together with data necessary to authenticate the log (e.g. cryptographic keys).

In use case /UC 250/ a single event log is composed of two log entries (one created by Computing Node X, the other one by Computing Node Y). Moreover, there are three log registries and only the VM$_{A1}$ log registry contains the whole event log.

7.6.2.2 Demo Storyboard

The following storyboard shows how Log Service can be used to spot infringements to the SLA, while preserving the privacy of all the actors involved. In particular, we have a User (Alice) who requires to always have a physical machine wholly dedicated to her VM instances and we show that, in case of migration on an already busy Computing Node (Computing Node 2), the infringement is noticed by Alice. Moreover, Alice will notice the infringement, in respect of the privacy of the owner(s) of the VM instance running on Computing Node 2.

1. User Alice creates an instance of her VM$_{A1}$ on Computing Node 1.
2. A new log entry is created for the start of VM$_{A1}$ (cf. /UC 240/):
   (a) A log entry is added to the log of Computing Node 1.
   (b) A log entry is added to the log of VM$_{A1}$.
3. User Bob creates an instance of her VM$_{B1}$ on Computing Node 2.

4. A new log entry is created for the start of VM_{B1} (cf. /UC 240/):
    (a) A log entry is added to the log of Computing Node 2.
    (b) A log entry is added to the log of VM_{B1}.
5. VM_{A1} is migrated from Computing Node 1 to Computing Node 2.
6. New log entries are created for the migration of VM_{A1} (cf. /UC 250/):
    (a) A log entry (sent VM_{A1} from Computing Node 1 to Computing Node 2) is added to the log of Computing Node 1.
    (b) A log entry (sent VM_{A1} from Computing Node 1 to Computing Node 2) is added to the log of VM_{A1}.
    (c) A log entry (received VM_{A1} from Computing Node 1 to Computing Node 2) is added to the log of Computing Node 2.
    (d) A log entry (received VM_{A1} from Computing Node 1 to Computing Node 2) is added to the log of VM_{A1}.
7. Alice retrieves the log of VM_{A1} (cf. /UC 260/, normal flow) and, after checking the life cycle of the VM instance, she sees that the VM instance has been started and migrated.
8. Alice retrieves the log of cloud infrastructure (cf. /UC 260/, alternative flow) and notices that another VM instance is running on Computing Node 2, hence breaking her SLA. For privacy reasons, Alice will see that “another VM instance” is running, but not “Bob’s VM_{B1}” is.

7.6.3 Architecture

7.6.3.1 High-level Design

The high level architecture is depicted in Figure 7.14. The current architecture is based on the Schneier-Kelsey scheme [SK99, see also D2.1.1, Chapter 7, for an overview of the scheme], which provides the security features required by a logging system. Despite this scheme lacks the protection against particular attacks, namely *truncation attack* and *delayed deletion*, it has been used as a foundation by most of the subsequent secure logging systems, which use a similar structure but with different cryptographic primitives (mainly public key instead of symmetric key cryptography). Therefore we consider it as a good candidate for designing the preliminary architecture of Log Service.

7.6.3.1.1 Log Core

The core component of the Log Service. This corresponds to the trustworthy actor of the Schneier-Kelsey scheme. Its main feature is to maintain the cryptographic material necessary to the User to verify the integrity of log entries. Because of this, this is the most critical component in terms of security and it must be kept as minimal as possible.

The Log Core is accessed by Cloud Components to *initialize* the Schneier-Kelsey scheme (cf. Section 7.6.3.2.1). It also support a *notification* indicating when a new log entry has been created, useful to keep a consistent state with the Log Storage (cf. Section 7.6.3.2.2).

Finally the Log Core is directly accessed by the User for the integrity verification (cf. Section 7.6.3.2.3).

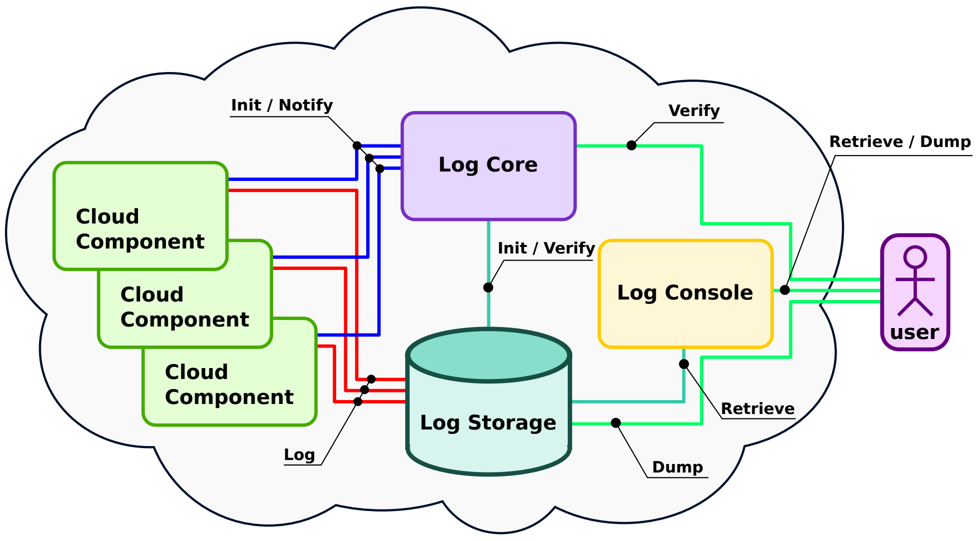


Figure 7.14: High-level architecture of Log Service.

7.6.3.1.2 Log Storage

The storage component. This is a resilient storage that provides the necessary availability of logs stored. This component has been separated from the Log Core to keep the critical part of the Log Service as minimal as possible. The Log Storage does not need to provide confidentiality or integrity of data since they are guaranteed by design, i.e. log entries are immediately protected when created.

This component will be built upon a resilient storage such as the resilient object storage described in Section 8.3. However, Log Storage should also provide functionality such as indexing, search or access control lists on log entries.

Log Storage provides the following functionality to the Cloud actors:

- *log:* allows a Cloud Component to store a new log entry (cf. Section 7.6.3.2.2).
- *download logs:* allows the User to download a (usually large) list of log entries (cf. Section 7.6.3.2.4).

Moreover, Log Storage is accessed by the Log Core and the Log Console to *retrieve* log entries (cf. Section 7.6.3.2.3).

7.6.3.1.3 Log Console

The enhancement to the Management Console to support log visualization. This is the main entry point for the User and allows to:

- *retrieve logs:* visualize a (relatively short) list of log entries (cf. Section 7.6.3.2.3).
- *dump logs:* download a (usually large) list of log entries (cf. Section 7.6.3.2.4). This triggers the creation of an archive on Log Storage and returns to the User the location of this archive.

In both cases the User must validate the integrity of the log entries, relying on the Log Core.

7.6.3.2 Sequence Diagrams

The following sequence diagrams describe the interactions between the components of Log Service. After the generic sequence diagrams, we discuss how to implement the use cases /UC 240/ and /UC 260/. The functions used in the sequence diagrams come from the API discussed in Section 7.6.4.

7.6.3.2.1 Init (Figure 7.15)

Before starting using Log Service it is necessary to initialize a new log registry. This operation must be done only once for each registry. To do this, the Cloud Component contacts the Log Core which will contact the Log Storage to create the appropriate log registry. The Log Core replies to the Cloud Component indicating the URI of the log registry.

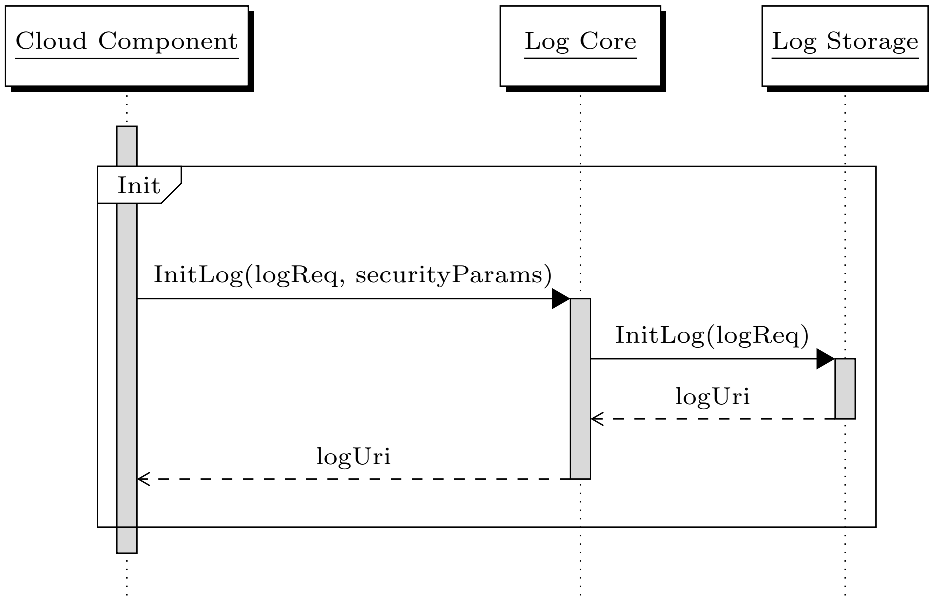


Figure 7.15: Sequence diagram for Log Service init.

7.6.3.2.2 Log (Figure 7.16)

Log Service provides a functionality for creating log entries. The Cloud Component locally creates a log entry[5]and then sends it to the Log Storage. The Log Storage response indicates whether the log entry was saved successfully. One single notification can be sent for multiple log entries, to reduce the network load.

Moreover, to keep a consistent state between Log Core and Log Storage, the Cloud Component notifies the Log Core that a new log entry was successfully created.

7.6.3.2.3 Retrieve and Verify (Figure 7.17)

Log Service provides functionality for retrieving and verifying log entries. The User requests the desired log entries to the Log Console which in turn contacts the Log Storage to retrieve them. Log Console sends the log entries back to the User.

The User can then access the Log Core to verify the integrity of logs. To reduce the network load, the Log Core directly retrieves the necessary log entries from the Log Storage.

[5]According to the Schneier-Kelsey scheme, the log entry must be protected to ensure integrity. This is done using the `libsklog`, described in Section 7.6.3.3.1.

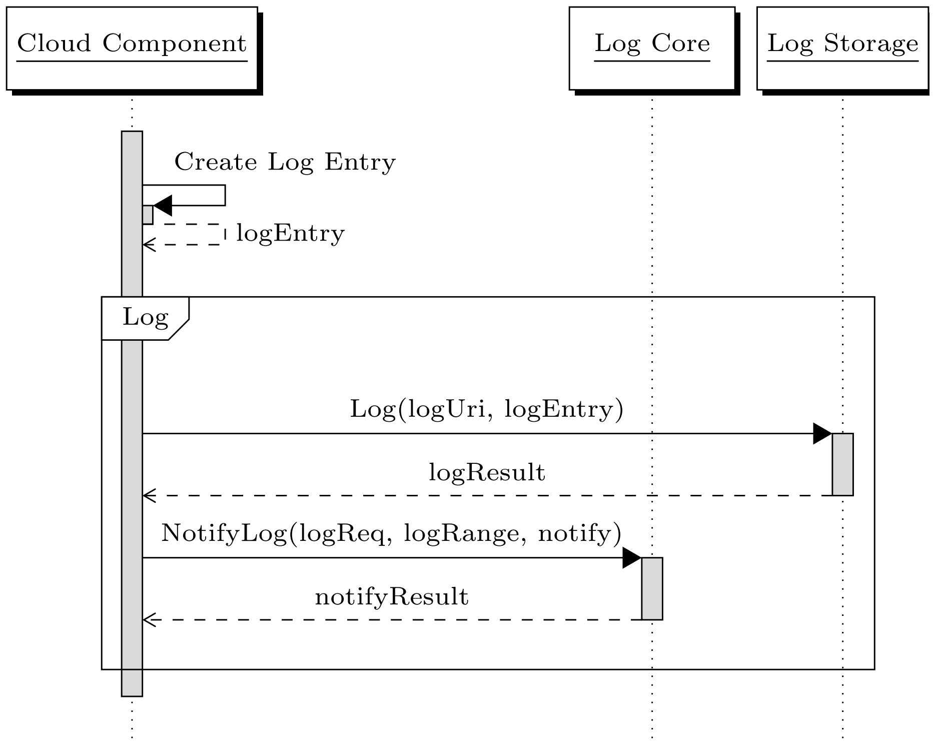


Figure 7.16: Sequence diagram for log entry creation.

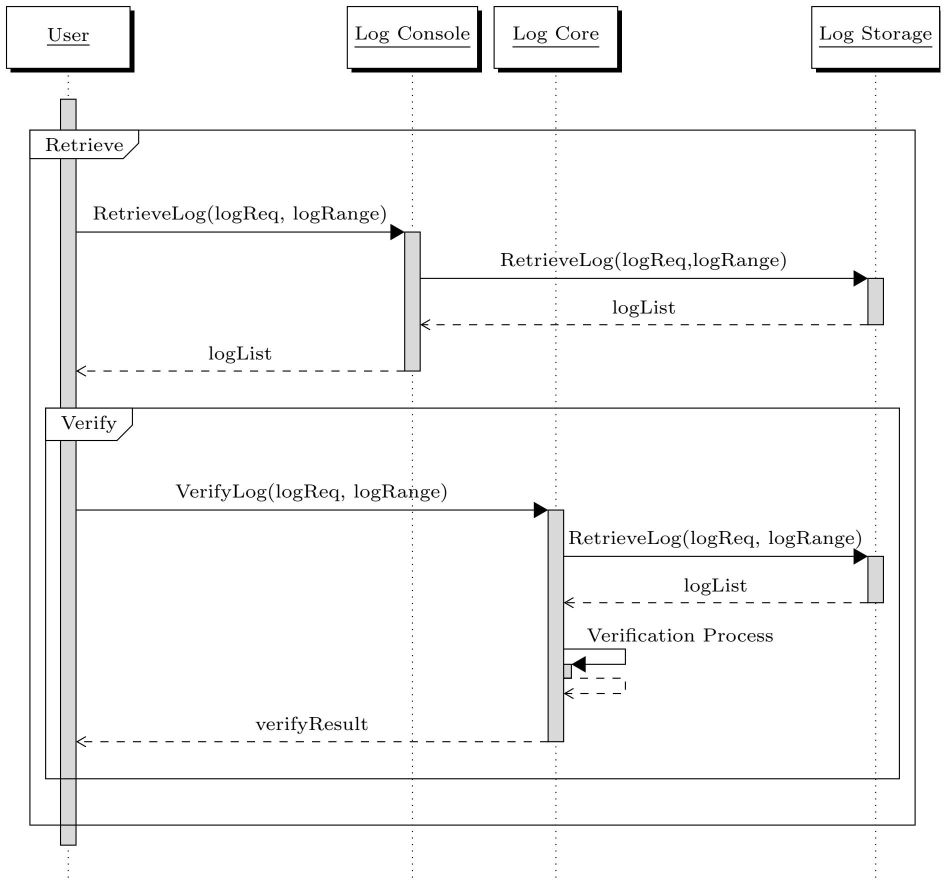

User
Log Console
Log Core
Log Storage
Retrieve
RetrieveLog(logReq, logRange)
RetrieveLog(logReq,logRange)
logList
logList
Verify
VerifyLog(logReq, logRange)
RetrieveLog(logReq, logRange)
logList
Verification Process
verifyResult

Figure 7.17: Sequence diagram for retrieve and verity log entries.

7.6.3.2.4 Dump and Download (Figure 7.18)

Log Service provides a functionality for retrieving a large quantity of log entries, called log dump. The User requests the log dump to the Log Console which in turn trigger the creation of an archive of logs on the Log Storage. The Log Console also indicates to the User the location from where he will download the archive. The User can poll the Log Console to know whether the archive creation process has finished.

When the Log Storage has finished to create the archive, it makes it possible for the User to start the download. After successful retrieval of the log dump, the User should perform a verification of the logs integrity.

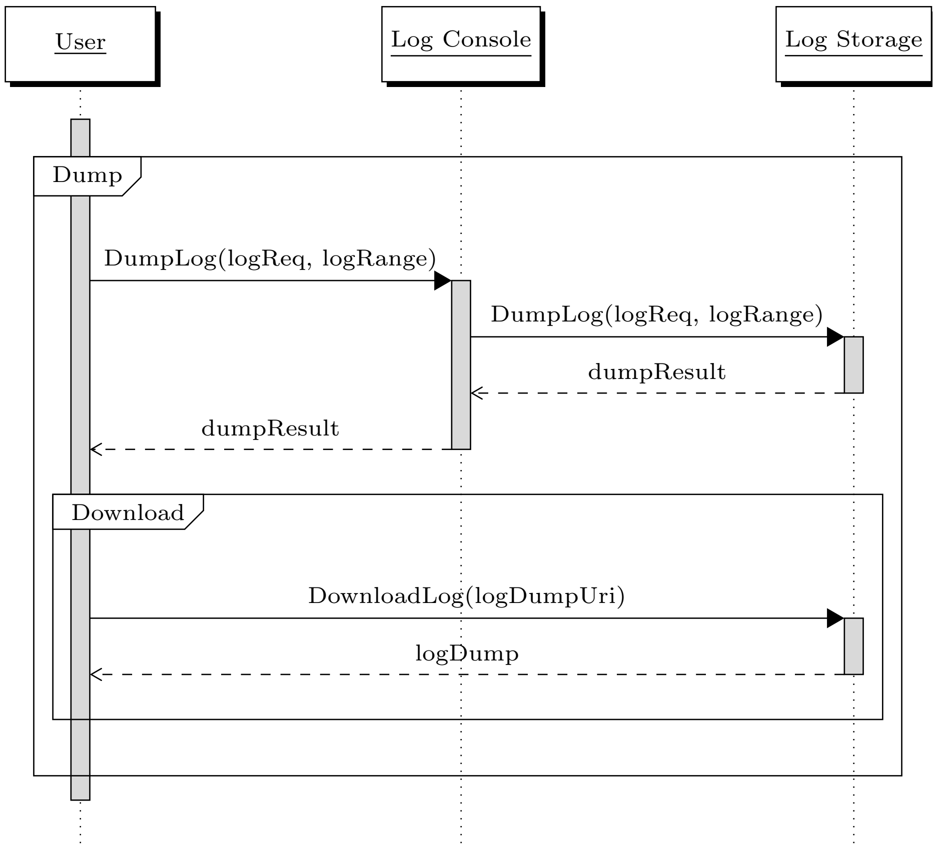


Figure 7.18: Sequence diagram for dumping large logs.

7.6.3.2.5 Selected Use Cases Implementation

7.6.3.2.5.1 Create event log 'Start Instance' /UC 240/. This use case uses the sequence diagram "Log" 7.6.3.2.1 to log the event. Before using this sequence diagram, it is necessary to use the sequence diagram "Init" 7.6.3.2.2 for initializing a new log registry. Note that the initialization is required only once.

7.6.3.2.5.2 Retrieve logs /UC 260/. This use case uses the sequence diagram "Retrieve and Verify" 7.6.3.2.2. Note that this use case implies not only "retrieving" the list of log entries from the Log Console, but also "verifying" the integrity of log entries with the Log Core.

7.6.3.3 Low-level Design

A preliminary low level architecture is depicted in Figure 7.19. This section anticipates some of the internals of a Computing Node that uses the Log Service.

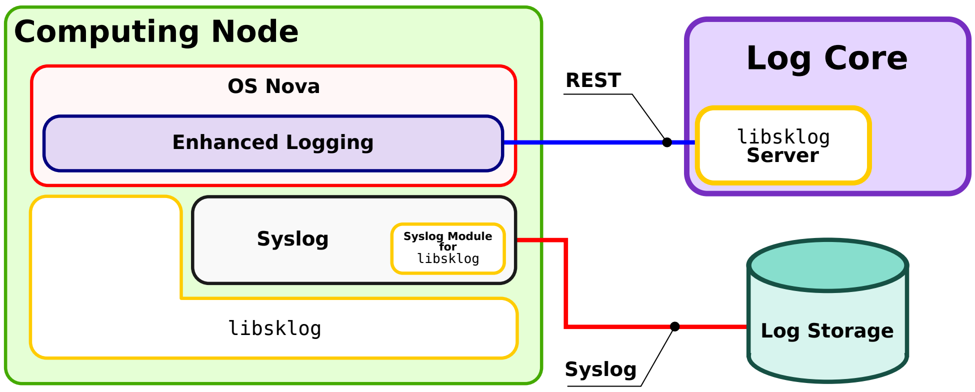


Figure 7.19: Low-level architecture of Log Service: internals of Computing Node.

7.6.3.3.1 `libsklog`

C library implementing the Schneier-Kelsey scheme. This library takes as input the string of a log to be saved and creates a log entry protected according to the Schneier-Kelsey scheme.

7.6.3.3.2 Syslog Module for `libsklog`

A module for `rsyslog` that integrates the functionality provided by `libsklog`. This module transforms a standard string sent to syslog into a log entry (as defined by the Schneier-Kelsey scheme), using `libsklog`. The log entry is then encoded as Base64 buffer and forwarded to syslog for standard handling.

7.6.3.3.3 Enhanced Logging in OS Nova

Enhancement of OS Nova to use the Python logging library, that provides a more detailed logging wherever necessary. In details, this enhancement provides a logging functionality that implements the Schneier-Kelsey scheme and an advanced access control mechanism.

7.6.4 API

Here the API of the components defined in the high level architecture is discussed, while the parameters and return values are described in Section 7.6.4.4.

7.6.4.1 Log Core

`InitLog()`

```
logUri ⇐ InitLog (logReq, securityParams)
```

Description.
The `InitLog` API is called by the Cloud Components to initialize a new logging session.

`NotifyLog()`

```
notifyResult ⇐ NotifyLog (logReq, logRange, notify)
```

Description.
The `NotifyLog` API is called by the Cloud Components to notify the Log Core of the writing of one or more new log entries.

`VerifyLog()`

```
verifyResult ⇐ VerifyLog (logReq, logRange, securityParams)
```

Description.
The `VerifyLog` API is called by the Users to verify the integrity of a range of previously stored log entries.

7.6.4.2 Log Storage

`InitLog()`

```
logUri ⇐ InitLog (logReq)
```

Description.
The `InitLog` API is called by the Log Core during the initialization of a new logging session.

`Log()`

```
logResult ⇐ Log (logUri, logEntry)
```

Description.
The `Log` API is called by the Cloud Components when a new event has to be logged.

`RetrieveLog()`

```
logList ⇐ RetrieveLog (logReq, logRange)
```

Description.
The `RetrieveLog` API is called by the Log Console when a User wants to retrieve some log entries or by the Log Core when a User needs to verify a range of log entries.

`DumpLog()`

```
dumpResult ⇐ DumpLog (logReq, logRange)
```

Description.
The `DumpLog` API is called by the Log Console during the dump process. The return value can be the progress of the dumping process or a `logDumpUri`.

`DownloadLog()`

```
logDump ⇐ DownloadLog (logDumpUri)
```

Description.
The `DownloadLog` API is called by the Users to download a compressed archive which contains a huge range of log entries.

7.6.4.3 Log Console

`RetrieveLog()`

```
logList ⇐ RetrieveLog (logReq, logRange)
```

Description.
The `RetrieveLog` API is called by the Users to retrieve a list of previously stored log entries.

`DumpLog()`

```
dumpResult ⇐ DumpLog (logReq, logRange)
```

Description.
The `DumpLog` API is called by the Users to request a huge list of previously stored log entries.

`AnalyzeLog()`

```
analyzeResult ⇐ AnalyzeLog (logReq, logRange, function, functionParams)
```

Description.
The `AnalyzeLog` API is called by the Users to request the execution of certain kind of analysis about a range of log entries.

7.6.4.4 API Parameters and Return Values

Parameter	Description
`function`	Specifies the function which has to be used to execute the analysis of a range of log entries
`functionParams`	Includes one or more parameters for the function used to perform the analysis of the log entries
`logEntry`	Includes the log entry priority and a string which describes the event to log
`logRange`	Defines a range of log entries
`logReq`	Includes a log registry identifier and some authorization credentials
`logUri`	Specifies the URI of the remote log entries collector (Log Storage)
`notify`	Specifies the event which has to be notified
`securityParams`	Includes some security elements (e.g. session keys)

Return Values	Description
`analyzeResult`	Represents the result related to the performed analysis
`dumpResult`	Represents the progress of the dumping process or the URI for dump downloading
`logDump`	Is a compressed archive which contains a huge number of log entries.
`logDumpUri`	Is the URI which has to be used for the dump downloading.
`logList`	Is a list of log entries.
`logResult`	Is a boolean value which represents the result of the Log procedure (success or failure)
`logUri`	Is the URI on where the remote logging has to be executed
`notifyResult`	Is a boolean value which represents the result of the NotifyLog() function.
`verifyResult`	Represents the result of the VerifyLog() function; it contains a boolean values and, only if the verification success, a pool of keys which can be used to decrypt a range of log entries

Chapter 8

Cloud of Clouds Middleware for Adaptive Resilience (WP 2.2)

Improved Availability and Resilience

8.1 State Machine Replication

Authors:
Alysson Bessani, Miguel Correia, Marcelo Pasin (FFCUL)

8.1.1 Overview

To build trustworthy clouds, security measures are clearly necessary to prevent malicious intrusions. But considering the enormous complexity of cloud infrastructures, platforms and services, it is unlikely that 100% of the programming errors (that lead to vulnerabilities) of such systems will be ever corrected. Any secure system can be deceived by exploiting its known defects, so measures that allow for *tolerating intrusions* must also be addressed when building the trustworthy clouds. To cope with this problem FFCUL is providing a state machine replication library that ensures *integrity* and *availability* of replicated services as long as at most a fraction of the replicas (usually less than a third) are compromised.

8.1.1.1 Description

Server and client are the basic structures used to implement distributed systems as clouds. The server offers services and the client uses such services by invoking them. An invocation is done by sending a request message from the client to the server, which returns the corresponding results as a reply message to the client.

Fault-tolerant distributed systems are implemented by replicating the components prone to failures and making them process message in a coordinated way, so they can fail independently without compromising the service availability and integrity. An intrusion-tolerant system is commonly modeled as a fault-tolerant system, capable of defending itself against *Byzantine failures*, in which a component is allowed to fail in arbitrary ways, including the most common stop and crash failures, but also processing requests incorrectly, corrupting their local state, or producing incorrect or inconsistent outputs.

Byzantine fault-tolerant services are implemented using *replicated state-machines*, that upon receiving a request deterministically change to a new state and send a reply. All state-machine replicas start with the same state and requests are sent to them using reliable, ordered, broadcasts from clients. The clients wait for a quorum of replies from different replicas and extract the result that a majority of them produced, ignoring thus results from faulty replicas. The Section 8.1.3 gives more details on how this is actually done.

8.1.1.2 Goals (Security, Privacy, Resilience)

The system will ensure the following properties:

- **Availability** by exploiting replication and diversity to run the replicas of the service on several clouds, thus allowing access to it as long as a subset of them is reachable.
- **Integrity** of the service executed as long as the majority (at least) of the clouds are correct and run the correct service code.

8.1.1.3 Required External Components

The State Machine Replication system is essentially middleware. It requires a set of individual replicas running the same software (typically four, to tolerate a single fault or intrusion) hosted on different operating systems and clouds in order to increase diversity and avoid common model faults (e.g., the same vulnerability is exploited in replicas).

8.1.1.4 Relationship with Activity 3

The system can be used to run critical services in multiple clouds. It is able to protect their availability and integrity even if some of the clouds used are offline or compromised.

8.1.2 Requirements

8.1.2.1 Selected Usecases

In this section we present some selected use cases for the state machine replication component. As a final note, we would like to remark that the general character of the use cases are a direct consequence of the generality of the components (i.e., in theory it can be used to provide fault tolerance to any deterministic service).

USE CASE UNIQUE ID	**/UC 270/ (Invoke)**
DESCRIPTION	User invokes an operation on a service
ACTORS	User
PRECONDITIONS	User client-side library knows enough service replicas
NORMAL FLOW	1. Client-side library sends a request to the server-side part of the library 2. Service libraries agree on request order 3. Service libraries deliver the request 4. State-machine processes the request and replies to the User 5. Service libraries deliver replies 6. Client library vote on the result of the request 7. Client library deliver result to the user code

USE CASE UNIQUE ID	**/UC 280/ (Start replicated service)**
DESCRIPTION	Starts several replicas for a replicated service
ACTORS	Project manager or an automated replica manager
PRECONDITIONS	The actor has privileges for the operation (not a common user)
POSTCONDITIONS	The replicated service exists
NORMAL FLOW	1. Set up an initial group (configuration file with IP and port of each replica) plus their public and private keys 2. Start multiple processes running the service (can be done by requesting image deployments) 3. Replica processes establish connection among them

USE CASE UNIQUE ID	**/UC 290/ (Add a Replica to a Replicated Service)**
DESCRIPTION	Starts a replica for a replicated service and adds it to the group
ACTORS	Project manager or an automated replica manager
PRECONDITIONS	The replicated service exists
PRECONDITIONS	The actor has privileges for the operation (not a common user)
POSTCONDITIONS	A new replica is up, running, and belongs to the group
NORMAL FLOW	1. Start a new process running the replicated service (can be done by requesting an image deployment) 2. Add a replica to the replicated service (issue an invoke to the replicas asking to join the group). This operation can only be made by clients with special privileges

USE CASE UNIQUE ID	**/UC 300/ (Remove a Replica from a Replicated Service)**
DESCRIPTION	Stops a replica of a replicated service and removes it from the group
ACTORS	Project manager or an automated replica manager
PRECONDITIONS	The replicated service exists
PRECONDITIONS	The actor has privileges for the operation (not a common user)
POSTCONDITIONS	A replica is appropriately shut down, after leaving the group
NORMAL FLOW	1. Remove a replica from the replicated service (issue an invoke to the replicas asking to leave the group). This operation can only be made by clients with special privileges. 2. Stop the process running the replicated service (can be done followed by an image shutdown)

8.1.2.2 [Optional] Non-functional Requirements

The effectiveness of any fault- and intrusion-tolerant solution requires a deployment that minimizes the probability of correlated failures. To achieve this goal for a Byzantine fault-tolerant (BFT) state machine replication service, we expect the system replicas to be deployed on different operating systems, Java virtual machines and hypervisors to avoid common-mode software faults and shared vulnerabilities and, additionally, to be hosted in different clouds (or, at least, different availability zones of the same cloud) to ensure provider-related outages and security related events do not affect more than one replica.

8.1.2.3 Demo Storyboard

1. Start a replicated service (for example, an in-memory key-value storage)
2. Start X clients that use the service every Y seconds
3. Manually kill a replica
4. Manually restart a replica, showing that it recovers its correct state
5. Trigger a malicious replica
   - Stop a good replica and start a malicious one
   - Malice can be done in the protocol or with a corrupted state
6. Show that the service keeps working
7. *(optional)* Replace the malicious replica with a good one
8. *(optional)* Show that the service keeps working

8.1.3 Design

8.1.3.1 Architecture

Byzantine Fault Tolerant (BFT) services are commonly implemented using replicated state-machines. Figure 8.1 shows a state machine that upon receiving (1) a message deterministically changes (2) to a new state and sends (3) a reply. All state-machine replicas start with the same state and requests are sent to them using reliable, ordered, broadcasts from clients. Majority (voting) is used within the clients to select the correct reply among those from all replicas. It has been shown that to tolerate f byzantine faults, one usually needs $3f + 1$ replicas, but it can be done with only $2f + 1$ replicas, for instance if they have a local trusted component to sign the exchanged messages.

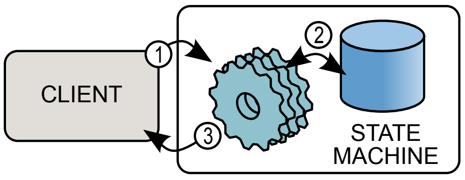


Figure 8.1: State machine in action

Figure 8.2 illustrates the behaviour of byzantine fault-tolerant replicated state machines in its main use case: *service invocation*. It starts (a) when a client program issues a request (1) which is sent by a client library to the service libraries (2). The service libraries agree (3), using any suitable protocol, upon the order in which messages are delivered to the state machines. The call to the service (b) then happens between the service library and the state machines. Finally (c), the service libraries send (1) each replica's replies to the client library, which collects (2) replies and vote, returning (3) the result to the client program.

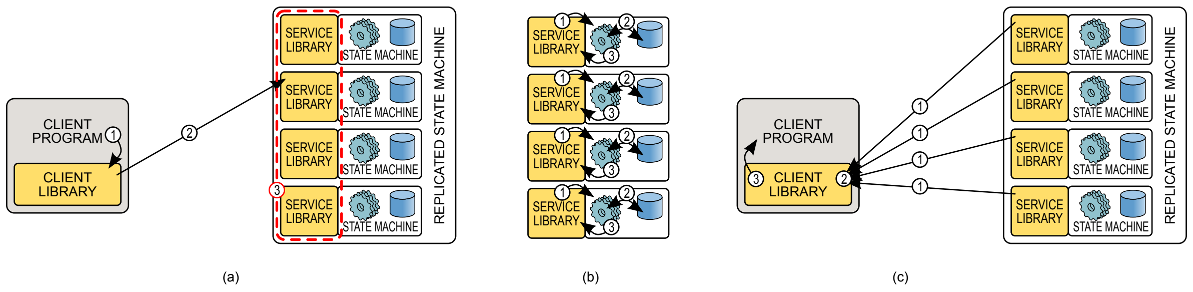


Figure 8.2: Replicated state machines

Prior to invoking a service, the client library must establish a means for addressing messages to the service libraries, which boils down to finding out the replicas addresses. It can be done in different ways, from a runtime variable or a file to a trusted configuration service containing the replicas addresses.

In order to get the replicas running, the main management operations are *deployment* and *shutdown*. They are implemented using the cloud PaaS deployment services existing. An initial deployment must be done, with a starting group of replicas. Later, single-replica deployments and shutdowns can be done individually. For doing so, the entity adding or removing replicas must also invoke the current replicas to *join* or to *leave* a replica group. None of these operations can be invoked by regular clients, only by the project manager or a by a trusted replica management service.

8.1.3.2 Sequence diagrams

Figure 8.3 describes how a message is processed in a service made fault tolerant using our BFT state machine replication.

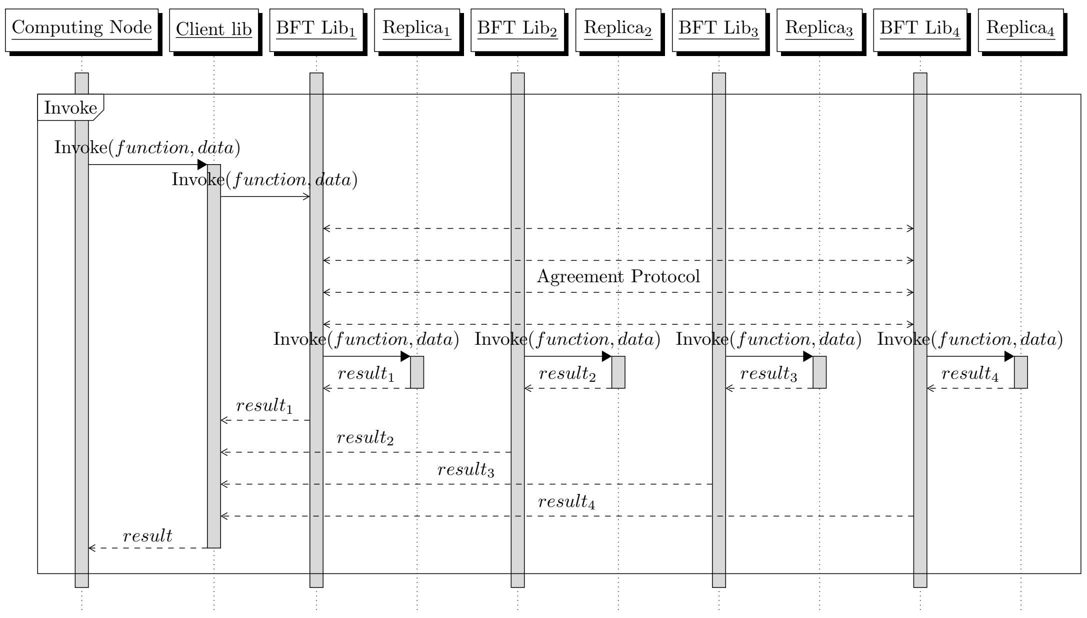


Figure 8.3: Sequence diagram for an Invoke call.

8.1.4 Implementation

The component defined in this section is being implemented in Java as an open-source programming library called BFT-SMART[1].

8.1.4.1 API

At client side, the client needs to create a `ServiceProxy` object with a constructor with the following signature:

```
ServiceProxy(int processId, String configHome)
```

In this constructor it is informed the process Id of the client (should be unique, and ideally associated with a public key available to the servers, if two-way authentication is used) and the local directory containing the configuration files. After that, there is only one simple operation:

```
byte[] invoke(byte[] command, boolean readOnly)
```

[1]Available at http://code.google.com/p/bft-smart/.

The client gives a byte array with a command and inform the library if this command is read-only or not. If a request is read-only the replication library will tentatively execute it without running a consensus among the replicas to establish total order.

At the server side, there is also a constructor of the `ServiceReplica` abstract class that needs to be extended by the service to be replicated. The parameters are exactly like the service proxy.

```
ServiceReplica(int processId, String configHome)
```

The class is abstract, and there are at least four operations that need to be implemented by each replicated service.

```
byte[] executeUnordered(byte[] command,
                        MessageContext ctx)

byte[] executeOrdered(byte[] command,
                      MessageContext ctx)

ServiceState getState()

void setState(ServiceState state)
```

The **executeUnordered** method is called by the replication library when a read-only message is delivered. The **executeOrdered** is a similar method called for processing normal requests (that are invoked with read-only = false). The replication library assigns timestamps, nonces, and statistics to each processed request and this information are available in an message context object. The **getState** and **setState** calls are used by the library to save and restore the state of the replica. These methods are fundamental for implementing replica recovery (after a failure) and dynamic reconfigurations (joining replicas need to obtain their states from other correct replicas).

8.2 Fault-tolerant Workflow Execution (FT-BPEL)

Authors:
Johannes Behl, Klaus Stengel (FAU)

8.2.1 Overview

8.2.1.1 Description

FAU will contribute with a PaaS infrastructure permitting the fault-tolerant execution of business processes in particular and workflows in general which are based on and composed of Web services. The infrastructure will be based on *BPEL*[2], an XML-based language for describing such workflows.

In order to provide a highly available fault-tolerant BPEL infrastructure, the FT-BPEL subsystem comprises a group of replicated BPEL engines which execute business processes described in the BPEL language. In the context of BPEL, business processes are expressed as a procedure which combines several Web services fulfilling single tasks. Since the business processes depend on the Web services they use, all of these Web services also have to be replicated. Furthermore, within the presented system ZooKeeper is used by replicas of the replication groups in order to conduct the necessary coordination.

8.2.1.2 Goals (Security, Privacy, Resilience)

- Availability, reliability and integrity in the presence of arbitrary faults

  Description: Business processes are usually critical tasks. If a business process could not be executed, crashes while it is executed or produces incorrect results for whatever reason, normally money would be lost and reputation would be damaged. Same holds for workflows in general, which can, for instance, provide crucial tasks within a Cloud infrastructure such as setting up VMs or carrying out maintenance work. Therefore, outages of business processes have to be avoided as much as possible.

  Techniques/research problems: By tolerating Byzantine, that is, arbitrary failures on the basis of state machine replication, high degrees of availability, reliability and integrity of Web-service-based workflows shall be achieved. In doing so, all used mechanisms should be as little invasive as possible to current BPEL infrastructures, on which the solution will be based on. Furthermore, existing Cloud services shall be used whenever suitable and the entire subsystem shall be highly configurable and adaptable.

  Assumptions: Web services used have to be deterministic.

8.2.1.3 Required External Components

- External coordination service

  The provided subsystem will make use of an external coordination service, namely Apache ZooKeeper[3].

[2]See "Web Services Business Process Execution Language Version 2.0" http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
[3]http://zookeeper.apache.org/.

8.2.1.4 Relationship with Activity3

The provided infrastructure can be used by A3 to execute crucial business processes or workflows in order to ensure high availability, reliability and integrity. The provided subsystem could be an alternative approach to traditional solutions based on, for example, Java servlets.

8.2.2 Requirements

8.2.2.1 Selected Use Cases

USE CASE UNIQUE ID	**/UC 310/ (Start VM instances)**
DESCRIPTION	A user starts all VM instances which will be part of the fault-tolerant BPEL infrastructure. It is possible to create the instances within different Clouds.
ACTORS	User and Clouds $C_1 \ldots C_f$.
PRECONDITIONS	User created the VM instances (cf. /UC 10/) on each Cloud C_i.
POSTCONDITIONS	The VM instances are running on Clouds $C_1 \ldots C_f$ and their IP addresses are known.
NORMAL FLOW	1. The images distributed to the different Clouds contain all necessary software packages needed to act as ZooKeeper, BPEL or service replica or as a client. However, the actual role of the VM instance will be only determined at the set-up stage (see /UC 320/). 2. The user starts the VM instances based on the distributed image. 3. Since the IP addresses of the instances are needed within the set-up stage, beforehand configured addresses are assigned to the started VM instances. 4. All VM instances are started and are prepared to run the different parts of the fault-tolerant BPEL infrastructure.
ALTERNATIVE FLOW (OTHER WAYS TO CONNECT STARTED VM INSTANCES)	1. Instead of configuring the IP addresses of the VM instances beforehand, they could be dynamically assigned while the starting procedure. In this case, some kind of mechanism has to be provided which can be used to obtain the IP addresses of the started instances.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Stop, reboot or terminate one or more VM instance, for instance to simulate crashes.

USE CASE UNIQUE ID	**/UC 320/ (Set up a fault-tolerant BPEL infrastructure)**
DESCRIPTION	A user sets up all parts of a fault-tolerant BPEL infrastructure including replicas of the BPEL engines, replicas of a ZooKeeper service, replicas of the services used by the business process and a client which also acts as the configurator of the test installation.
ACTORS	User and Clouds $C_1 \dots C_f$.
PRECONDITIONS	Prepared VM instances are running on Clouds $C_1 \dots C_f$ and their IP addresses are known.
POSTCONDITIONS	Each VM instance hosts either a ZooKeeper, a BPEL or service replica and the whole system is ready for usage.
NORMAL FLOW	1. One VM instance is selected to act as client and also as configurator. 2. It is ensured that the client instance has access to a list containing the IP addresses of all other VM instances (e. g. by means of a simple text file). 3. Set-up scripts are executed on the client in order to initialize all other VM instances. During this process a role is assigned to each instance. That is, on each VM instance either a ZooKeeper, a BPEL or a service replica is started. Furthermore, all instances are provided with the necessary configuration data needed to connect the corresponding replication group.

USE CASE UNIQUE ID	**/UC 330/ (Retrieve the number of active replicas)**
DESCRIPTION	A user retrieves the numbers of all active replicas. The total number comprises the numbers of BPEL, ZooKeeper and service replicas.
ACTORS	User, client VM instance
PRECONDITIONS	A BPEL infrastructure was set up and is ready for usage.
POSTCONDITIONS	—
NORMAL FLOW	1. The user gets the number of all replicas currently active within the BPEL infrastructure by means of the client. 2. The client connects to the other VM instances and collects their current status. 3. The numbers are presented to the user.

USE CASE UNIQUE ID	**/UC 340/ (Execute a normal business process)**
DESCRIPTION	A user starts a business process by using the client VM instance. The process is carried out by the BPEL infrastructure without problems and the correct results are returned to the user.
ACTORS	User, client VM instance
PRECONDITIONS	A BPEL infrastructure was set up and is ready for usage.
POSTCONDITIONS	The BPEL infrastructure is still ready for usage.
NORMAL FLOW	1. The user starts a simple business process by means of the client. 2. The business process is carried out by the BPEL infrastructure. 3. The correct result of the business process is presented to the user.

USE CASE UNIQUE ID	**/UC 350/ (Execute a business process in the presence of a outage)**
DESCRIPTION	A user starts a business process by using the client VM instance. While the process is carried out by the BPEL infrastructure a crash of one BPEL replica is simulated by terminating the corresponding VM instance. Nevertheless, the correct results are returned to the user.
ACTORS	User, client VM instance
PRECONDITIONS	A BPEL infrastructure was set up and is ready for usage.
POSTCONDITIONS	The BPEL infrastructure is still ready for usage.
NORMAL FLOW	1. The user starts a simple business process by means of the client. 2. While the business process is carried out by the BPEL infrastructure, the VM instance of one BPEL replica is terminated. 3. Nevertheless, the correct results of the business process are presented to the user.
ALTERNATIVE FLOW (SIMILAR USE CASES)	1. Instead of terminating the VM instance of a BPEL replica, an instance of a ZooKeeper or service replica could be terminated.

8.2.2.2 Demo Storyboard

The intention of this story is to show how business processes can be carried out even in the presence of outages. For that purpose a FT-BPEL infrastructure is built up comprising a replicated BPEL engine, one single replicated Web service and a ZooKeeper service which utilizes replication as well. Within this story, the management of the infrastructure is carried out by a single client. The client is responsible for the configuration of the infrastructure, as well for starting requests and presenting the results to the user.

1. A user creates several VM instances distributed over different Clouds (cf. /UC 310/).
2. After one VM instances has been selected as the client, this client is used to set up the BPEL infrastructure (cf. /UC 320/).
3. In order to show the status of the built up infrastructure, the number of all active components can be retrieved at any time (cf. /UC 330/).
4. The execution of business processes can be started by means of the client (cf. /UC 340/). It is possible to terminate single VM instances while a business process is being executed. Such simulated outages of single components are tolerated by the infrastructure, so that the user gets correct results even in this situation.

8.2.3 Architecture

8.2.3.1 High-level Design

As stated before, BPEL is a language for describing business processes. These business processes are not only composed of Web services but they are provided as Web services themselves. This is done by so-called *BPEL engines* responsible for executing *process definitions* written in BPEL. In that way, clients can invoke business processes in the same manner as they would invoke customary Web services. Called by a client, the BPEL engines carry out the appropriate process, whose definition has been imported before. In particular, the engines invoke all Web services necessary to fulfill the request.

Existing BPEL engines usually log changes of state during the execution in order to tolerate crashes. However, this solution has several drawbacks: The execution is slowed down significantly, the business process is not available during recovery and the solution depends on a reliable storage. Furthermore, the reliability of the Web services used has not been addressed at all by existing implementations.

FT-BPEL addresses these problems by actively replicating not only the BPEL engines but (optionally) also the Web services in a combined architecture. As depicted in Figure 8.4 a transformation process is installed which transparently prepares process definitions before they are imported into the engines. The transformation process is mainly used to redirect invocations to proxies responsible for the replication. Theses proxies make use of an external Apache ZooKeeper service for coordination, dynamic retrieval of system information, configuration, crash detection and request ordering.

8.2.3.2 Sequence Diagrams

Figure 8.5 shows the sequences of messages exchanged while setting up a demo system and processing a single request.

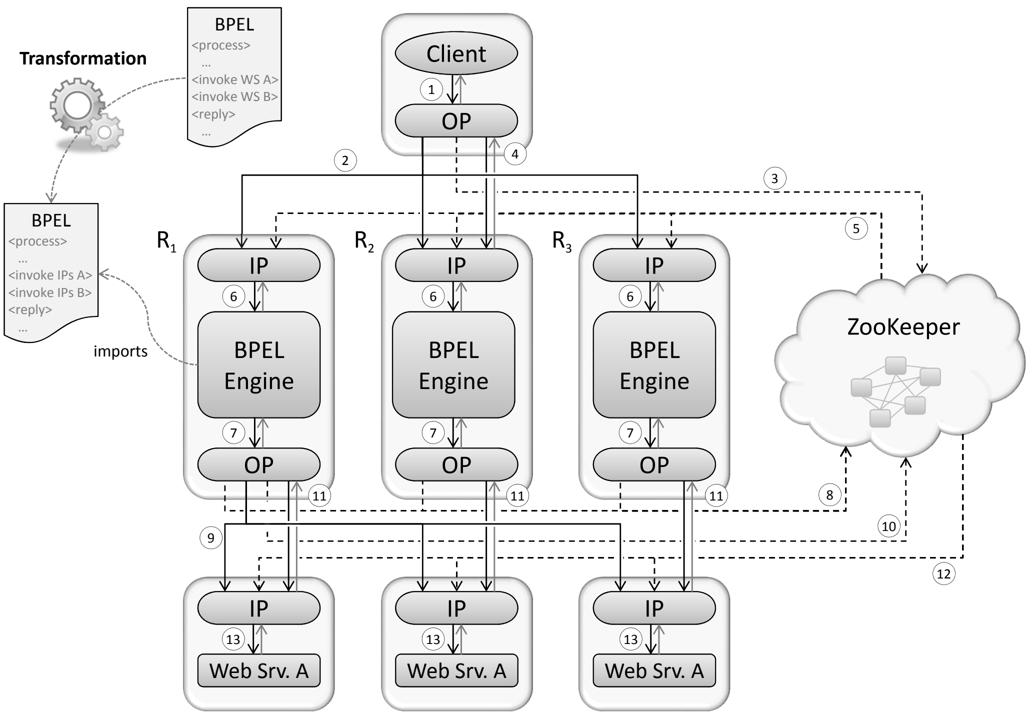


Figure 8.4: Architecture of FT-BPEL

For the set-up process, the client acts as coordinator. When the system shall be set up, it reads the IP addresses of all other computing nodes eventually executing the replicas, either of the replicated BPEL engine, the Web services of ZooKeeper. First, the ZooKeeper replicas are initialized. For that purpose, the client sends the IP addresses of all computing nodes dedicated for executing ZooKeeper replicas to each of these nodes. After the replicas are up and running they send an acknowledgment to the client. Now, the client is able to store further configuration data within ZooKeeper. This way, when the BPEL and the Web service replicas are initialized by the client, by sending the ZooKeeper connection endpoints to the computing nodes, they can retrieve necessary information from ZooKeeper.

The second part of Figure 8.5 depicts the sequence of a normal request processing. In that case, the client is nothing more than a usual client invoking services. This is done by firstly sending the data of a request to all BPEL replicas. Then, the request has to be registered in ZooKeeper to obtain a global ordering of requests. ZooKeeper informs the BPEL replicas about the request ordering via callbacks registered before. Subsequently, the BPEL replicas elect a new leader, if necessary, responsible for invoking the Web services. Invocation of Web services is analog to the invocation of BPEL engines. If all steps of a business workflow are processes, a reply is sent to the client.

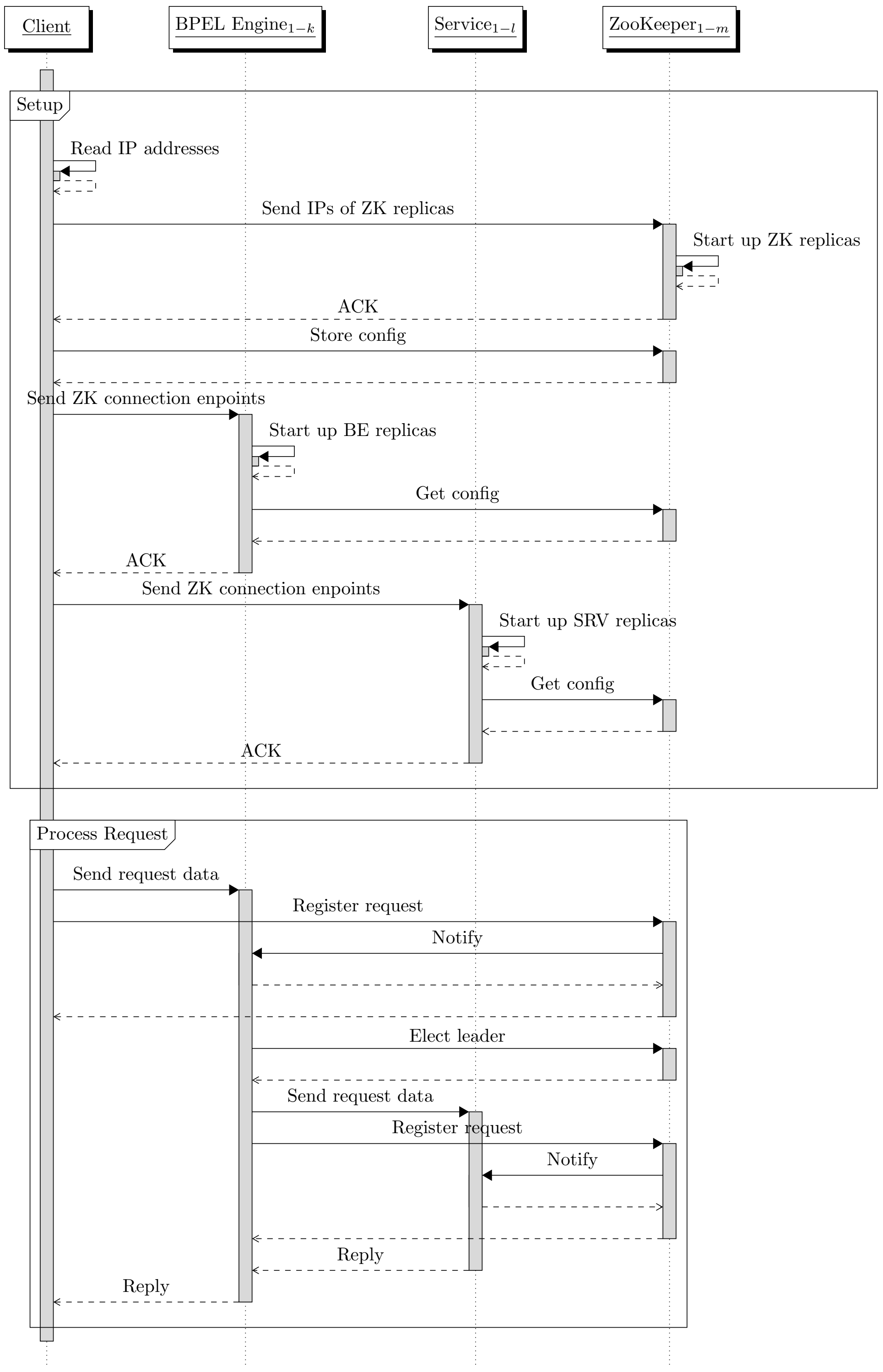


Figure 8.5: Sequence diagrams for FT-BPEL

8.2.4 API

In the following, the APIs and protocols used by FT-BPEL are presented in groups according to three different views of the system: The *internal* view describes the internal behavior of the system, especially the incoming and outgoing communication of the proxies. The *service provider* utilizes FT-BPEL to provide high available, fault-tolerant Web services or to provide a platform for the reliable execution of business processes as PaaS. *Clients*, in turn, use the services offered by the service provider.

Internal Proxies communicate between each other over a TCP-based protocol, make use of ZooKeeper via its Java library and interact with the BPEL engines by means of SOAP. Since these are internal protocols, encapsulated from service providers and clients, they are not further specified and potentially subject to change.

Service Provider Service provider describe business processes by means of BPEL[4] and import the definitions into the system. Here, it has to be noted, that currently only deterministic BPEL processes are supported. In particular, BPEL flows must no be used, yet. Furthermore, service providers can configure the system via provided tools. How this is done in detail is not decided yet, but tools will come with a short manual describing their usage.

Client Usually, clients communicate with Web services with the help of libraries. These libraries have to be adjusted if FT-BPEL is used. Currently, only implementations of JAX-WS 2.x[5], a JAVA API normally used for this purpose, is supported. Clients have to be configured, so that they use the adjusted FT-BPEL implementation of this API. However, this affects only the configuration phase of the clients, the actual Web service invocation does not change, at least not from the viewpoint of the clients.

[4]The BPEL API specification can be found here: `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[5]See `http://jax-ws.java.net/`.

Resilient Storage

8.3 Resilient Object Storage

Authors:
Sören Bleikertz, Christian Cachin, Thomas Groß, Michael Osborne (IBM)
Alysson Bessani, Miguel Correia, Marcelo Pasin (FFCUL)

8.3.1 Overview

8.3.1.1 Description

The object model for cloud storage has become extremely popular, after its introduction with Amazon's Simple Storage Service (S3) in 2006. It allows reads and writes of simple blobs, each one identified by a unique name (also called a "key"). A multitude of commercial providers offer such *blob storage* services today.

In collaboration with FFCUL, we will contribute a system that builds reliable and secure storage through a federation of object storage services from multiple providers. Multiple clients may concurrently access the same remote storage provider and operate on the same objects. They do this through an interface that contains the basic and most common operations of object cloud storage. (Since every vendor provides the same basic operations but slightly different advanced operations, the system only uses the common denominator of all providers.)

The software is a library run by each client before it accesses cloud storage; the management and setup is the same as for accessing one storage provider, and the library does not require client-to-client communication. The library requires some cryptographic credentials (public keys) of all clients to be present.

The storage system provides confidentiality through encryption, integrity through cryptographic data authentication, and reliability through data replication and erasure coding. Key management for encryption and authentication keys is integrated.

8.3.1.2 Goals (Security, Privacy, Resilience)

The system ensures the following security/resilience properties:

- **Availability:** Through exploiting replication and diversity to store the data on several clouds, it allows access to the data as long as a subset (generally, a large enough majority) of them is reachable.

- **Integrity:** Data can be retrieved correctly even if some of the clouds corrupt data, lose it, or adversarially manipulate it. The system builds on so-called Byzantine fault-tolerant replication that stores data on several providers.

- **Confidentiality:** By encrypting the stored data, it protects the confidentiality of the data against disclosure to one or more cloud providers. The system may use a novel secret sharing scheme, whereby encryption keys are maintained collaboratively by a (sufficiently large) majority of the cloud providers. No (small enough) faulty minority can learn anything about the stored data, not even by colluding.

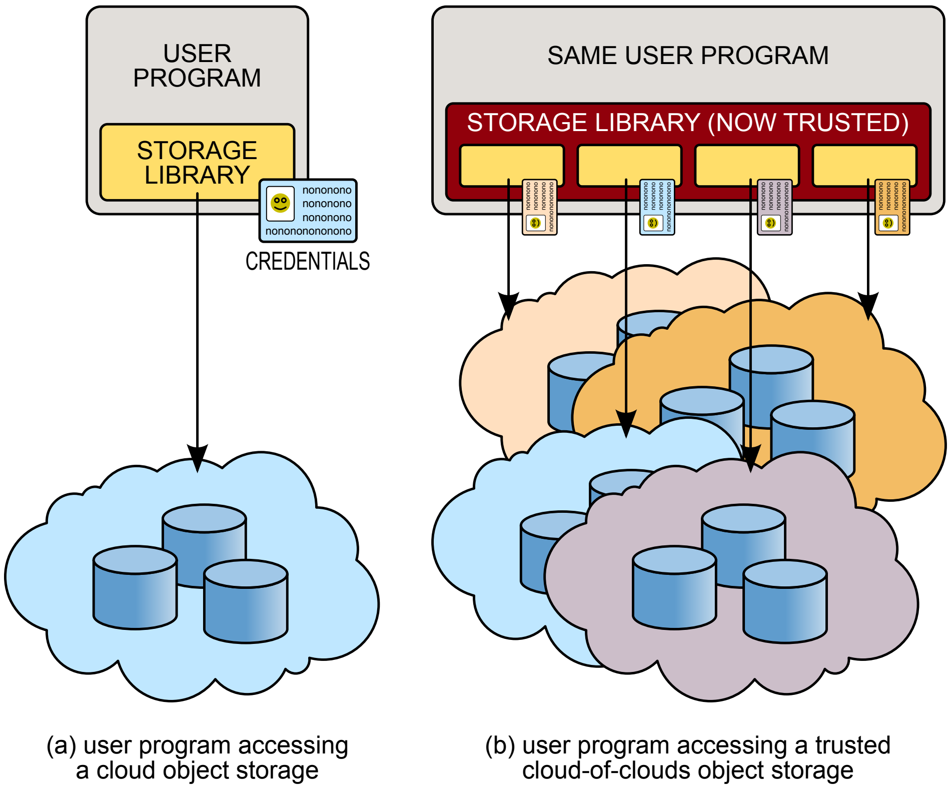

USER PROGRAM
STORAGE LIBRARY
CREDENTIALS
SAME USER PROGRAM
STORAGE LIBRARY (NOW TRUSTED)
(a) user program accessing a cloud object storage
(b) user program accessing a trusted cloud-of-clouds object storage

Figure 8.6: Client software accessing (a) a legacy object storage and (b) a cloud-of-clouds trusted object storage.

8.3.1.3 Required External Components

At its basic form, the system is essentially a middleware, requiring a set of individual clouds (typically four or more). It uses JClouds (`http://www.jclouds.org`) or a library provided by Amazon for accessing Amazon S3 (`http://aws.amazon.com/documentation/s3/`).

We are currently discussing the possibility of providing it as a proxy server offering a dependable object storage service to a set of clients inside a private cloud. At this point it is still not clear which interface this proxy should support: S3, CloudStack (`http://www.openstack.org/projects/storage/`) or OGF/SNIA CDMI (Cloud Data Management Interface - `http://www.snia.org/cdmi`).

8.3.1.4 Relationship with Activity 3

The system can be used to store data that is critical in terms of availability, integrity and confidentiality. Moreover, this data can be shared by multiple (trusted) parties using the untrusted clouds as coordination media.

8.3.2 Requirements

8.3.2.1 Selected Usecases

USE CASE UNIQUE ID	**/UC 360/ (Write Object)**
DESCRIPTION	A user A writes an object to the object storage
ACTORS	User
PRECONDITIONS	Local file contains an object
PRECONDITIONS	User has a number of credential (several clouds)
POSTCONDITIONS	Object is stored encrypted, with parts spread an replicated on several clouds
NORMAL FLOW	1. User collects a number of credentials for multiple object storage systems (e.g., S3) 2. User writes object X 3. The library linked to the user code transparently encrypts and splits X 4. Encrypted object is stored on multiple object storages

USE CASE UNIQUE ID	**/UC 370/ (Read Object)**
DESCRIPTION	A user B (different from user A) reads an object from the object storage
ACTORS	User
POSTCONDITIONS	Object is stored encrypted, with parts spread and replicated on several clouds
PRECONDITIONS	User has the credential for those clouds
POSTCONDITIONS	Local file contains the object
NORMAL FLOW	1. User sees object X on an object storage 2. User reads object X 3. Object is readable in a local file

USE CASE UNIQUE ID	/UC 380/ (Delete Object)
DESCRIPTION	A user C (different from users A and B) removes a object from the object storage
ACTORS	User
POSTCONDITIONS	Object is stored encrypted, with parts spread an replicated on several clouds
PRECONDITIONS	User has the credential for those clouds
POSTCONDITIONS	Object is no longer in the clouds
NORMAL FLOW	1. User sees object X on a object storage 2. User deletes object X 3. Object disappears!

8.3.2.2 Demo Storyboard

1. Data objects are stored through the object store interface.
2. Clients repeatedly write an object.
3. Clients repeatedly read an object.
4. No single storage cloud and no small enough coalition of storage clouds may violate the availability, integrity, and confidentiality guarantees applied to the stored objects.

8.3.3 Design

8.3.3.1 Architecture

A *container* or a *bucket* groups one or more objects together. An *object* is a file of arbitrary length. An *ACL* (access control list) contains information about who can read/write in an object or a bucket. *Authorization* is a credential that authenticates the requester.

8.3.3.2 Sequence diagrams

8.3.4 Implementation

The implementation provides a library that can be accessed by applications according to the *jclouds blobstore* interface, which is available at `http://code.google.com/p/jclouds/` and with source code from `git://github.com/jclouds/jclouds.git`.

8.3.4.1 API

The following interface is offered by the component. The API represents a subset of the jclouds blobstore interface.

```
/**
 * Synchronous access to a BlobStore such as Amazon S3
 */
```

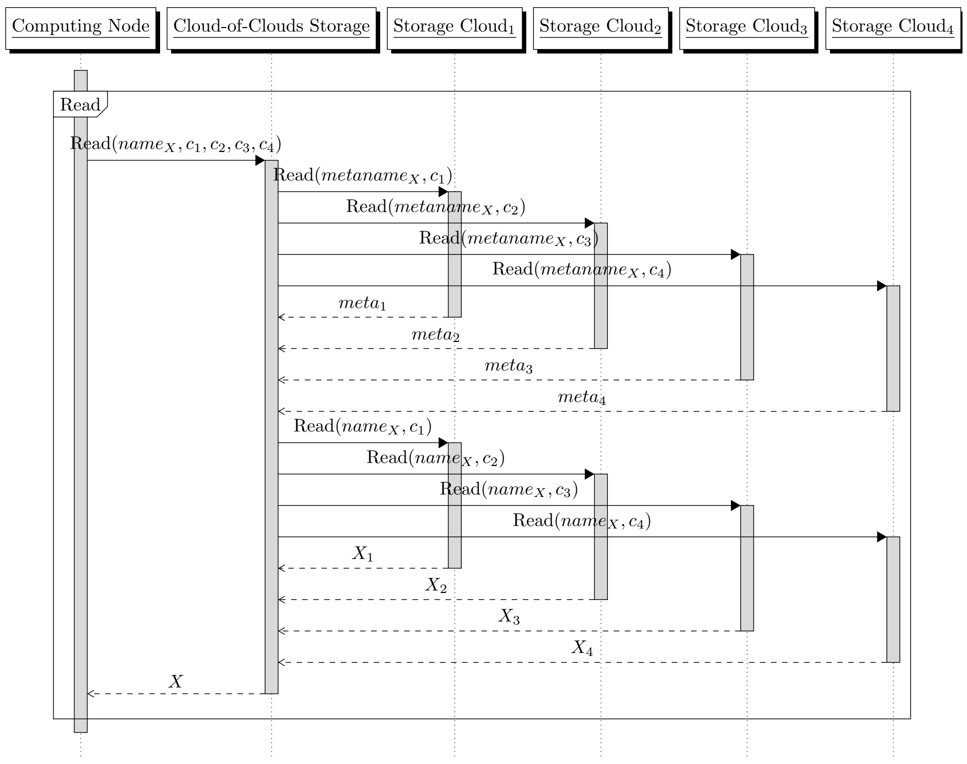


Figure 8.7: Sequence diagram for an Read call.

```
public interface BlobStore {
   /**
    * determines if a service-level container exists
    */
   boolean containerExists(String container);

   /**
    * Creates a namespace for your blobs
    *
    * @param location
    *            some blobstores allow you to specify a location, such as US
      -EAST, for where this
    *            container will exist. null will choose a default location
    * @param container
    *            namespace. Typically constrained to lowercase alpha-numeric
       and hyphens.
    * @return true if the container was created, false if it already
      existed.
    */
   boolean createContainerInLocation(@Nullable Location location, String
      container);

   /**
    * Lists all resources in a container non-recursive.
    *
    * @param container
    *            what to list
    * @return a list that may be incomplete, depending on whether PageSet#
      getNextMarker is set
    */
```

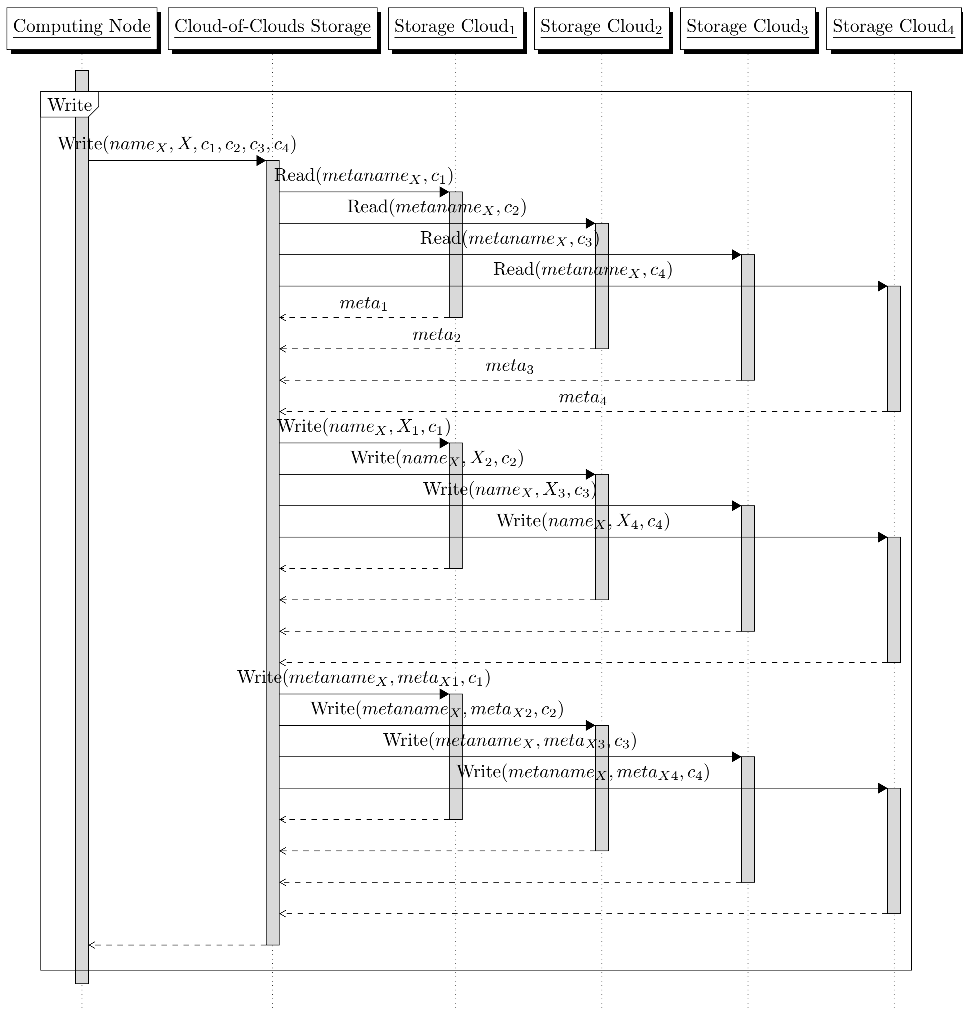

Computing Node
Cloud-of-Clouds Storage
Storage Cloud$_1$
Storage Cloud$_2$
Storage Cloud$_3$
Storage Cloud$_4$
Write
Write($name_X, X, c_1, c_2, c_3, c_4$)
Read($metaname_X, c_1$)
Read($metaname_X, c_2$)
Read($metaname_X, c_3$)
Read($metaname_X, c_4$)
$meta_1$
$meta_2$
$meta_3$
$meta_4$
Write($name_X, X_1, c_1$)
Write($name_X, X_2, c_2$)
Write($name_X, X_3, c_3$)
Write($name_X, X_4, c_4$)
Write($metaname_X, meta_{X1}, c_1$)
Write($metaname_X, meta_{X2}, c_2$)
Write($metaname_X, meta_{X3}, c_3$)
Write($metaname_X, meta_{X4}, c_4$)

Figure 8.8: Sequence diagram for an Write call.

```
    PageSet<? extends StorageMetadata> list(String container);

    /**
     * This will delete the contents of a container at its root path without
         deleting the container
     *
     * @param container
     *            what to clear
     */
    void clearContainer(String container);

    /**
     * This will delete everything inside a container recursively.
     *
     * @param container
     *            what to delete
     */
    void deleteContainer(String container);

    /**
     * Determines if a blob exists
     *
     * @param container
     *            container where the blob resides
     * @param directory
     *            full path to the blob
     */
    boolean blobExists(String container, String name);

    /**
     * Adds a {@code Blob} representing the data at location {@code
       container/blob.metadata.name}
     *
     * @param container
     *            container to place the blob.
     * @param blob
     *            fully qualified name relative to the container.
     * @return etag of the blob you uploaded, possibly null where etags are
       unsupported
     * @throws ContainerNotFoundException
     *             if the container doesn't exist
     */
    String putBlob(String container, Blob blob);

    /**
     * Retrieves a {@code Blob} representing the data at location {@code
       container/name}
     *
     * @param container
     *            container where this exists.
     * @param name
     *            fully qualified name relative to the container.
     * @return the blob you intended to receive or null, if it doesn't exist
       .
     * @throws ContainerNotFoundException
     *             if the container doesn't exist
     */
    Blob getBlob(String container, String name);

    /**
     * Deletes a {@code Blob} representing the data at location {@code
       container/name}
     *
     * @param container
     *            container where this exists.
     * @param name
     *            fully qualified name relative to the container.
     * @throws ContainerNotFoundException
     *             if the container doesn't exist
     */
    void removeBlob(String container, String name);

    /**
     * @return a count of all blobs in the container, excluding directory
       markers
     */
    long countBlobs(String container);

}
```

8.4 Confidentiality Proxy for S3

Authors:
Michael Gröne, Norbert Schirmer (SRX)

8.4.1 Overview

Integrating untrusted Amazon Simple Storage Service (Amazon S3 [amab]) -based storage into the trusted cloud infrastructure is another approach to reach resilient storage. Therefore the trusted cloud infrastructure needs a middleware component this section gives an overview of.

8.4.1.1 Description

SRX will contribute to the trusted cloud infrastructure with a confidentiality proxy for S3. The component is implemented as a security service which is part of the security kernel (cf. Deliverable D2.1.1, Chapter 12) and managed by TOM. It will transparently encrypt data of a mounted file system (Linux) according to a TVD cf. Deliverable D2.1.1, Chapter 12), and allows to integrate untrusted S3-based storage into the trusted cloud infrastructure (cf. Figure 8.9). The S3 proxy does not directly expose the S3 interface to the User. Instead the S3-based storage is mounted as a file system (via s3fs [s3f]). So a User stores and reads ordinary files through a Linux file system or (optional) a Server Message Block (SMB) share instead of accessing the bucket(s) directly, which would mean interaction with buckets and objects via the SOAP and REST API [amac]. The encryption happens transparently within the TrustedServer which attaches the S3-based storage as an encrypted file system to all VM instances belonging to a TVD. The encryption key is derived from the TVD of the VM instance and managed by TOM.

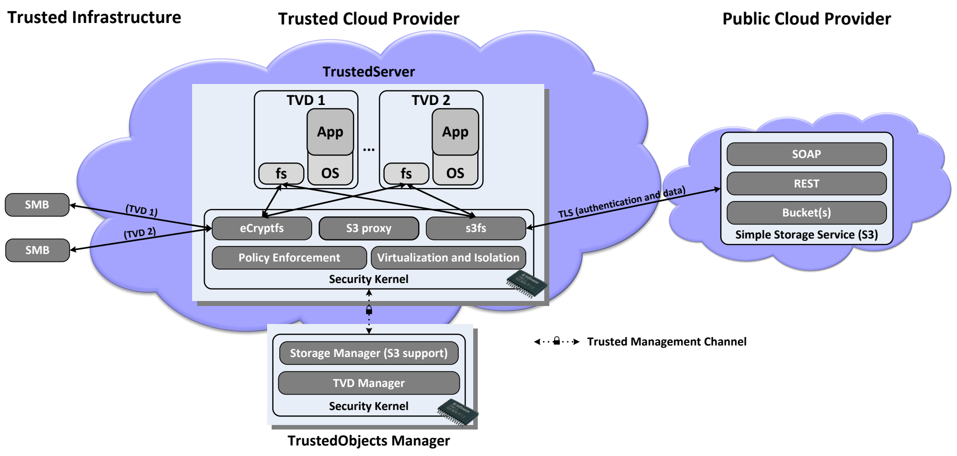


Figure 8.9: Overview of Confidentiality proxy for S3.

8.4.1.2 Goals (Security, Privacy, Resilience)

The system will provide confidentiality based on transparent TVD wide encryption. The key's are managed by TOM. The main purpose of this component is to demo an integrated prototype: TOM (including TVD Management), TrustedServer, untrusted (Public) Cloud. At a later point the component may be replaced by a more sophisticated storage component, e.g.'Resilient Object Storage' designed by partners IBM and FFCUL (cf. 8.3).

8.4.1.3 Required External Components

The system requires the following external components:

- S3-based storage component. For now this is Amazon S3, part of Amazon Web Services (AWS).
- a FUSE file system that allows mounting an Amazon S3 bucket as a local file system. For now this is s3fs [s3f] which stores files natively and transparently in S3.

8.4.1.4 Relationship with Activity3

The system can be used to store data that is critical in terms of integrity, confidentiality and privacy. Moreover, this data can be shared by multiple trusted parties if they belong to the same TVD using the untrusted cloud (Amazon AWS) as coordination media.

8.4.2 Requirements

This section gives an overview of the requirements for the S3 proxy component.

8.4.2.1 Preconditions

Requirements that have to be fulfilled already, because they are needed for the development process.

/PR 30/ TrustedServer component
A TrustedServer component is required (cf. section 7.5).

8.4.2.2 Execution Environment

This section specifies the minimal hardware, software and infrastructure requirements for the user to run our product successfully.

8.4.2.2.1 Hardware

- TPM 1.2 Platform

8.4.2.2.2 Software

- TrustedServer (cf. section 7.5)
- Operating System which supports an SMB file system and has network access to internal network infrastructure.

8.4.2.2.3 Infrastructure

- TrustedObjects Manager (cf. section 9.2)
- Trusted Management Channel (cf. section 9.3)
- access to an cloud storage infrastructure (e.g. Amazon S3). Generally these are valid S3 credentials, as today from AWS, and a (fast) internet connection.

8.4.2.3 Security Environment

This section describes the security aspects of the environment in which the product is intended to be used and the manner in which it is expected to be employed.

8.4.2.3.1 Assumptions A description of assumptions describe the security aspects of the environment in which the component will be used or is intended to be used.

/A 80/ Trusted Organization Administrator
The organization administrator of the managed IT infrastructure is non-malicious.

/A 90/ Correct hardware
The underlying hardware (e.g., CPU, devices, TPM) does not contain backdoors, is non-malicious, and behaves as specified.

/A 100/ Security of eCryptfs
Certain security objectives of the appliance and the file system are delegated to eCryptfs[6], the cryptograhpic file system layer we employ. Hence, the system relies on eCryptfs to achieve its security objectives in handling the necessary secrets, and controlling the encryption layers, so it does not contain backdoors, is non-malicious, and behaves as specified.

/A 110/ Trusted Domain
The internal network infrastructure the S3 proxy is connected to is a well controlled environment based on security policies and therefor a trusted domain.

8.4.2.4 Security Objectives

The security objectives address all of the security environment aspects identified. The security objectives reflect the stated intent and are suitable to counter all identified threats and cover all identified organizational security policies and assumptions.

8.4.2.4.1 Security Objectives of the S3 proxy The S3 proxy transparently encrypts S3-based storage with eCryptfs. The encryption/decryption key is only known by the TOM so the cloud provider (e.g. U.S.-based Amazon) only has data in encrypted form.

8.4.2.4.2 Security Objectives of the IT-Environment S3 proxy Integrity Prove The IT-environment provides a mechanism that allows the S3 proxy to convince remote parties and local users about its integrity. Common examples of such a mechanism are a secure bootstrap architecture as used, e.g., by the AEGIS architecture [SCG+03], or an authenticated bootstrap architecture as specified by the TCG [tpm].

8.4.2.4.3 Security Objectives for the S3 proxy

[6] https://launchpad.net/ecryptfs

/O 130/ S3 proxy Integrity

Using the functionalities offered by the IT-Environment, the S3 proxy should be able to prove to remote parties and local users that the integrity of the S3 proxy is not violated. Changes in the S3 proxy must be detectable by both the user and remote parties. Such changes can drastically affect the security properties of the system, and therefore mechanisms must be put in place to prevent entrusting sensitive data to such a compromised system.

/O 140/ Integrity of User Data

User data should remain integer during storage.

/O 150/ Confidentiality of User Data

User data should remain confidential during storage.

8.4.2.5 Security Requirements

This subsection defines the security requirements that have to be satisfied by the S3 proxy. The statements shall define the functional and assurance security requirements that the product and the supporting evidence for its evaluation need to satisfy in order to meet the security objectives.

/SR 60/ TPM protected encryption keys and credentials

The credentials material used for S3 authentication and encryption keys must be protected by a TPM. The confidentiality and integrity of user data relies on this assumption.

8.4.2.6 Selected Usecases

USE CASE UNIQUE ID	**/UC 390/ (Write File)**
DESCRIPTION	A user writes a file to the S3 file system
EXTENDS	/UC 80/
ACTORS	User, TrustedServer
PRECONDITIONS	VM instance in TVD$_A$ is running on TrustedServer, S3 is mounted into VM instance.
POSTCONDITIONS	File is stored encrypted on S3
NORMAL FLOW	1. User writes file X on a mounted S3 storage within a VM instance 2. TrustedServer transparently encrypts file X according to TVD policy 3. Encrypted file X is stored on S3

USE CASE UNIQUE ID	**/UC 400/ (Read File From TVD)**
DESCRIPTION	A User opens a file from the S3 file system (from the TVD the file was encrypted for)
EXTENDS	/UC 80/
ACTORS	User, TrustedServer
PRECONDITIONS	VM instance in TVD$_A$ is running on TrustedServer, S3 is mounted into VM instance, file X is stored on S3 and belongs to the same TVD.
POSTCONDITIONS	File is readable in plain text
NORMAL FLOW	1. User sees file X on a mounted S3 storage 2. User opens file X for reading 3. File X is readable in plain text
ALTERNATIVE FLOW	1. cf. /UC 410/

USE CASE UNIQUE ID	**/UC 410/ (Optional: Read File From Other TVD)**
DESCRIPTION	A User tries to open a file from S3 (from another TVD)
ACTORS	User, TrustedServer
PRECONDITIONS	Users VM instance in TVD$_B$ is running on Trusted-Server, S3 is mounted into Users VM instance, file X is stored on S3 and belongs to TVD$_A$.
POSTCONDITIONS	File $_X$ is not readable in plain text
NORMAL FLOW	1. Depending on the TVD policy the user can see the file name X or even the name is encrypted. 2. When user opens the file he can only see encrypted data

8.4.2.7 Demo Storyboard

The following story shows how the S3 proxy can be used to: (1) Write and read files in a TVD (in a public Cloud storage scenario (cf. /UC 80/)); (2) assist the User/ infrastructure to guarantee confidentiality/privacy of data if written to a file storage.

User Alice and Bob are assigned to different TVDs because of their roles in the organization. They are able to use the same storage system through a SMB share to save and read data. This data may contain sensitive information, related to privacy or confidentiality aspects. Alice and Bob have to trust the infrastructure that privacy and confidentiality concerns are always fulfilled. Both do not know where data is stored and if it is encrypted or not.

1. Files are stored through the standard file system (SMB) interface.

2. S3-based storage (objects, placed buckets) is mounted as a file system (via s3fs) into a TVD and transparently encrypted.
3. User Alice writes a file in a TVD$_A$ which is then encrypted transparently (cf. /UC 390/).
4. User Alice reads a file in a TVD$_A$ which is decrypted transparently (cf. /UC 400/).
5. User Bob reads a file in a TVD$_B$ which stays encrypted.
6. No storage cloud may violate the confidentiality/privacy guarantees applied to the stored files.

8.4.3 Architecture

8.4.3.1 High-level Design

The high-level architecture of the S3 proxy component is described here.

- Big picture and relations with other components:
  Is the component required by other Activity 2 components? No.
  Does the component require an other Activity 2 components? Yes. It requires
  - TOM (cf. 9.2) and Trusted Management Channel (cf. 9.3) for configuration.
  - TrustedServer (cf. 7.5) as a platform to be integrated into as a service within the SecurityKernel.
- Hints on implementation:
  Language: C, bash, C++.
  Existing SW: s3fs, eCryptfs, SMB protocol.

8.4.3.2 Sequence Diagrams

The following sequence diagrams describe the interactions between the (sub)components involved when using the S3 proxy as a security service as part of the secure hypervisor of an TrustedServer component. They implement the use cases /UC 390/ and /UC 400/ defined in section 8.4.2.6 and how to setup the S3 proxy. The functions used in the sequence diagrams come from the API discussed in section 8.4.4. The *TURAYA Manager* is the entry point in the Security Kernel which is in charge to apply the configurations and commands received by TOM. Preconditions are:

- Project Manager has added S3 configuration (AWS / S3 credentials) on TOM.
- Compartment and its configuration (in particular TVD and S3 configuration) is installed on a TrustedServer managed by TOM.

8.4.3.2.1 Setup (Figure 8.10). To setup the S3 proxy the TURAYA Manager service within the SecurityKernel create and mount s3fs. After this is done the S3 proxy cryptomounts it in eCryptfs. Then TURAYA Manager attaches the S3 to all TVD compartments as configured in TOM.

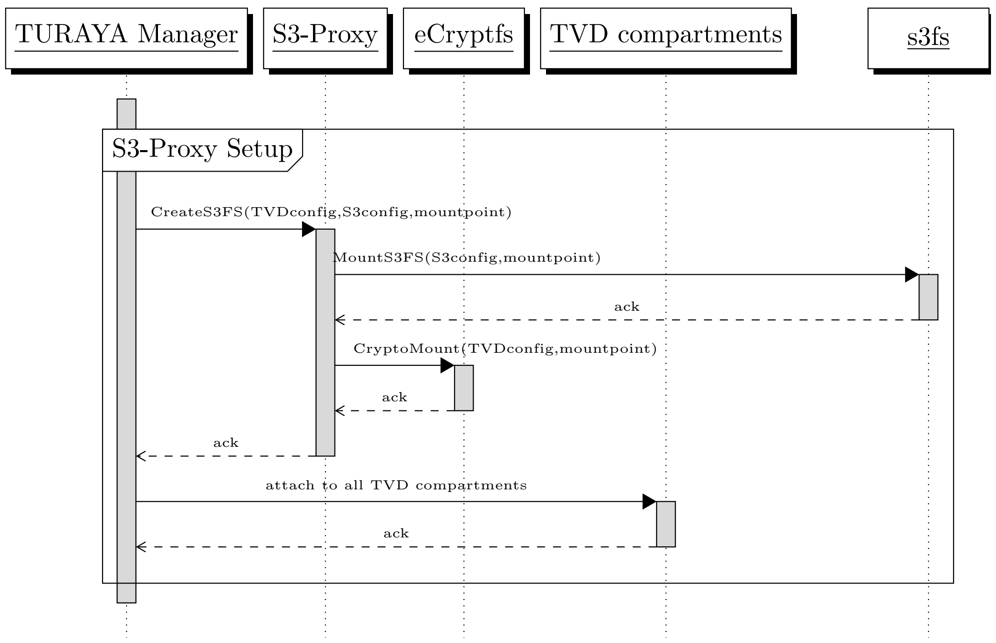

TURAYA Manager
S3-Proxy
eCryptfs
TVD compartments
s3fs
S3-Proxy Setup
CreateS3FS(TVDconfig,S3config,mountpoint)
MountS3FS(S3config,mountpoint)
ack
CryptoMount(TVDconfig,mountpoint)
ack
ack
attach to all TVD compartments
ack

Figure 8.10: Sequence diagram for setup of Confidentiality proxy for S3.

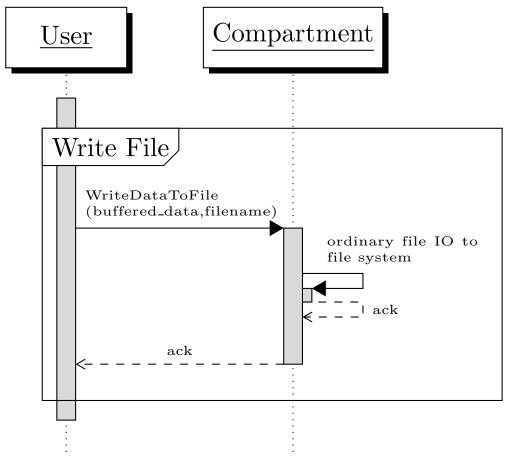

User
Compartment
Write File
WriteDataToFile (buffered_data,filename)
ordinary file IO to file system
ack
ack

Figure 8.11: Sequence diagram for /UC 390/ use case.

8.4.3.2.2 Write File (Figure 8.11). If a User saves data to a file in her actual compartment (and assigned TVD(s)) it is transparently encrypted and saved on S3.

8.4.3.2.3 Read File From TVD (Figure 8.12). If a User saves data to a file in her actual compartment (with TVD(s) assigned through TOM) it is transparently encrypted and saved on S3.

8.4.4 API

The API of the Confidentiality proxy for S3 is quite straightforward since it uses standardized file system (Linux) and SMB (optional) semantics and protocols on the one hand and s3fs on the other hand.

Public or private? Public.

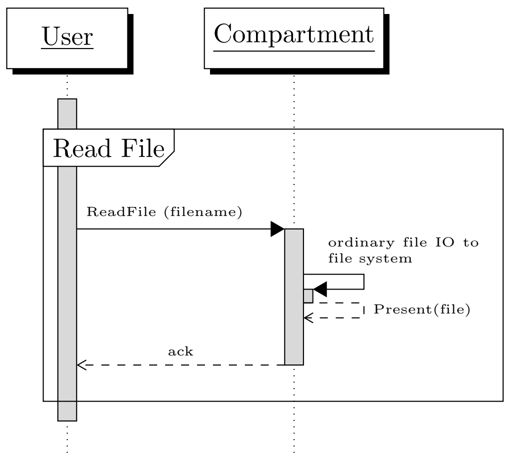


Figure 8.12: Sequence diagram for /UC 400/ use case.

Standard It will be mounted into the file system (Linux) within the VM instance.

Sketch of API Standard Linux file system and (optional) SMB file system [smb] semantic and file accesses on the one hand, s3fs [s3f] semantic on the other, with the limitations imposed by the Amazon S3 back-end / API [amac] (eventual consistency [eve]). For the public interface we just have the standard file system (Linux) interface and semantics. Setup and configuration of the component is done via GUI of TOM and processed internally as depicted in Figure 8.10.

Chapter 9

Cross-layer Security and Privacy Management (WP 2.3)

High-availability Management

9.1 Access Control as a Service (ACaaS)

Authors:
Imad M. Abadi (OXFD)

9.1.1 Overview

9.1.1.1 Description

The provision of automated management of clouds virtual resources is a fundamental requirement for the success of future cloud computing. Such automated management would require understanding the properties of cloud infrastructure and its policies, and it would also require understanding cloud user requirements. Cloud user requirements should be continually considered by cloud provider by matching user requirements and infrastructure properties in normal operations as well as during incidents. We are planning to develop an Enterprise Rights Management (ERM) tool, which we refer to as Access Control as a Service (ACaaS). In this section we describe one component of the ACaaS which is our planned contribution to TClouds project.

The objective of ACaaS is to act as a policy decision point to manage the hosting of VM instances at an appropriate Computing Node. Specifically, it extends /UC 10/ and /UC 100/ — ACaaS component verifies that a Computing Node satisfies User requirements when hosting its VM instance. This is achieved by matching cloud's User requirements and Computing Node infrastructure policy/properties which are as follows:

User Requirements (Dynamic Properties) — A cloud user interacts with the cloud provider via cloud webpage and supplied APIs. This enables users to define *user requirements*. User requirements include technical properties, QoS/SLA requirements (e.g. system availability, reliability measures, and lower/upper resource limits), and security and privacy requirements (e.g. location of data distribution and processing). How complex such requirements would be based on the type of user.

Infrastructure Properties (Static Properties) — clouds' physical infrastructure are very well organized and managed by multiple parties, e.g. Cloud Admin. Those people define the physical infrastructure properties which would cover: components reliability and connectivity, components distribution across cloud infrastructure (e.g. how far components are from each other), redundancy types, servers clustering and grouping, network speed, etc.

Infrastructure Policy — Policies are defined by authorized employees to control the management of cloud environment.

9.1.1.2 Goals (security/privacy/resilience)

- Provide a component that consider user requirements during normal operations as well as in incidents. Example of user requirements includes the following: enforce location restrictions, manage the hosting of dependent applications (e.g. group a set of applications to be hosted within physical proximity and, simultaneously ensure they do not run on the same Computing Node), and exclude certain physical properties from hosting a user application.

  Techniques/research problems We believe that establishing trust in the cloud starts by establishing trust in the clouds' operational management. This would require building trustworthy automated management services that can automatically manage clouds' infrastructure and consider security and privacy by design. ACaaS is the initial step in this direction.

  Assumptions We make the following assumptions:

  - Although, we do not assume that all cloud employees are trusted, we assume that cloud provider trust a partial set of employees who interact and manage the ACaaS.
  - We assume that moving physical Computing Node across physical locations is controlled by the set of trusted cloud Admin who are trusted to reflect such movement in the ACaaS.
  - We assume that the hardware of Computing Node are secure and trusted. We also require that they incorporate a Trusted Platform Module chip (TPM).

9.1.1.3 Required external components (relationship with Activity2)

- Resilient database management system.

  Name/description: A resilient database management system is required to store user requirements, infrastructure properties and policies.

  Features (security/privacy/resiliency): The database must be resilient to not to be a single point of failure.

  Required API (provided by the external component): SetUserProperties(), GetUserProperties(), SetInfrastructureProperties(), and GetInfrastructureProperties().

- Trusted platform.

  Name/description: ACaaS will rely on a trusted platform with a hardware root of trust.

  Features (security/privacy/resiliency): The trusted platform must be equipped with a hardware root of trust capable of securely store and use cryptographic keys. It should be capable of reporting its integrity.

  Required API (provided by the external component): jTSS

- Others

  Name/description Research at this subsystem is still at early stage and it is very likely that we will require other components that could be provided by WP2.1

9.1.1.4 Relationship with Activity3

Managing clouds' hosting environment based on real user requirements is one of the fundamental A3 security and privacy requirements.

9.1.2 Requirements

9.1.2.1 Selected Use Cases

USE CASE UNIQUE ID	**/UC 420/ (Set User Requirements)**
DESCRIPTION	User securely provisions his requirements to the ACaaS using a proper interface.
ACTORS	User
PRECONDITIONS	User has a set of requirements.
POSTCONDITIONS	Only ACaaS has access to User requirements.
NORMAL FLOW	1. User connects to the ACaaS using a provided GUI and then uploads his requirements by filling a form. 2. ACaaS securely stores User requirements into a database such that only a ACaaS can access them.

USE CASE UNIQUE ID	**/UC 430/ (Set Infrastructure Properties)**
DESCRIPTION	Cloud Admin securely provisions cloud's infrastructure properties and policies to the ACaaS using a proper interface. This process can be automated by using client agents running on Computing Node which collect such properties and pass them over to the ACaaS.
ACTORS	Cloud Admin. Client agents could also be involved.
PRECONDITIONS	Cloud Admin has the infrastructure properties and policies, or an agent collects such information on behalf of Cloud Admin and provides them to the ACaaS.
POSTCONDITIONS	Only ACaaS has access to infrastructure properties and policies.
NORMAL FLOW	1. Cloud Admin or an agent connects to the ACaaS using a provided GUI and then uploads the cloud's infrastructure properties and policies. 2. ACaaS securely stores the properties and policies into a database such that only a ACaaS can access them.

USE CASE UNIQUE ID	**/UC 440/ (Secure VM Creation)**
DESCRIPTION	This use case extends /UC 10/. Specifically, before creating a VM instance either ACaaS suggests an appropriate hosting Computing Node or Cloud Admin provides the Computing Node. In either case ACaaS verifies whether the Computing Node could satisfy User requirements.
ACTORS	Computing Node, User, and Management Component.
PRECONDITIONS	User set his requirements using /UC 420/. Cloud Admin set the infrastructure properties using /UC 430/.
POSTCONDITIONS	The new VM instance is hosted on a Computing Node that can satisfy User properties.
NORMAL FLOW	1. A User sends a requests to the Management Component to create a VM, as described in /UC 10/. 2. Cloud Admin using the Management Component could either suggest a Computing Node to host the VM instance or the ACaaS can suggest one. 3. The Management Component must coordinate with the ACaaS to ensure that the Computing Node satisfies User requirements. 4. If the Computing Node was provided by the Cloud Admin, ACaaS would then return true if the Computing Node satisfies user properties, and false otherwise. If no Computing Node has been provided, ACaaS would then suggest one. 5. Finally, the Management Component would proceed as discussed in /UC 10/.

USE CASE UNIQUE ID	**/UC 450/ (Secure VM Migration)**
DESCRIPTION	This use case extends /UC 100/ when migrating a VM instance from Computing Node X to Computing Node Y. Specifically, ACaaS could either suggest an alternative Computing Node Y that satisfies User requirements, or it can verifies whether Computing Node Y provided by Cloud Admin satisfies User requirements.
ACTORS	Computing Node, Cloud Admin, and Management Component.
PRECONDITIONS	User created a VM instance and set his requirements using /UC 420/. Cloud Admin set the infrastructure properties using /UC 430/. A VM instance is securely created using /UC 440/.
POSTCONDITIONS	The VM instance is migrated to a Computing Node that can satisfy User properties.
NORMAL FLOW	1. In this case a User VM is needed to be migrated to a new destination Computing Node, say Computing Node Y. This can be automated or explicitly instructed by a Cloud Admin. 2. If Computing Node Y is provided by Cloud Admin, the Management Component must first check with the ACaaS whether Computing Node Y satisfies User requirements. Alternatively, ACaaS would suggest a new destination Computing Node that can satisfies the User requirements. 3. In the latter case ACaaS would return to the Management Component the Computing Node that could host the VM instance. In the former case ACaaS would return true if Computing Node Y satisfies User requirements and false otherwise. 4. The Management Component would then proceed as discussed in /UC 100/.

9.1.2.2 Demo Storyboard

In this part we discuss a scenario using the above use cases on TClouds application context. We have the following assumptions:

- We assume that Cloud Admin has already defined the infrastructure properties as discussed in /UC 430/. One of these properties is the physical location of Computing Node. Cloud Admin uniquely identifies Computing Node using their endorsement keys.

- We assume that moving physical Computing Node across physical locations is controlled by the set of Cloud Admin who are trusted to reflect such movement in the ACaaS.

A hospital system defines their requirements using a provided GUI (see, for example, /UC 420/). One of the hospital requirements is to ensure that their data is processed in UK. Our scenario starts by a cloud insider who is not trusted and instructs the migration of the hospital VM instance to a different physical server outside UK. Now, our use case /UC 450/ should reject such a migration process. This is because the hosting Computing Node does not satisfy the defined User requirements.

9.1.3 Architecture

In this section we briefly outline our implementation architecture. We use OpenStack Compute as management framework. OpenStack Compute is composed of many components as illustrated in Figure 9.1. We mainly discuss the ones related to our framework (see [Opec] for detailed discussion about OpenStack components): *nova-api* intermediates the communications between OpenStack and cloud users, *nova-database* is the central repository for OpenStack management data, *nova-schedule* manages the hosting of VM instances at cloud physical layer, and *nova-compute* creates and terminates VMs.

We are planning to provide policy management by proposing mechanisms which matches user properties with infrastructure properties. In this case the policy will be enforced at client side by the calling agent. In our prototype we plan to use *nova-database* to securely store user properties and the structure of the cloud (i.e. physical properties). We build a relational database to hold, for example, the following: i) physical layer components, their infrastructural properties, their membership, and the policy governing them; ii) virtual layer components, associated user properties; and iii) management policies.

We are planning to provide two interfaces to interact with *nova-database* via *nova-api*: the first is related to managing users' properties and the second is for managing clouds infrastructural properties and policies. At this stage the infrastructure properties would be provided by administrators. These data should only be accessed and managed using cloud server agent running as part of *nova-schedule*. Future implementation will consider the utilization of cloud client agents running at Computing Node to collect such properties and securely push them to *nova-database*.

Nova-schedule is the central component of our scheme which controls the hosting of VMs at physical resources. Current implementations of *nova-schedule* do not consider the entire cloud infrastructure neither they consider the overall user and infrastructure properties.

The way the proposed component works is as follows (see Figure 9.2): the Cloud Admin add the infrastructure properties using a proper GUI. The infrastructure properties are stored inside *nova-database*. The user request a server he provides his requirements using a provided GUI. These are also stored using *nova-database*. Subsequently, a VM instance is created based on the provided user requirements. *Nova-compute* when starting a VM instance it sends a request to *nova-schedule*. *Nova-schedule* then decides on Computing Node to host the VM instance by considering both of the infrastructure properties and user requirements.

9.1.4 API

The proposed component API is composed of the following main functions (see Figure 9.2).

- int SetUserRequirements(String[] User Requirements) — Allow users to add their requirements in secure way. This is to be implemented in a form of a GUI, where users

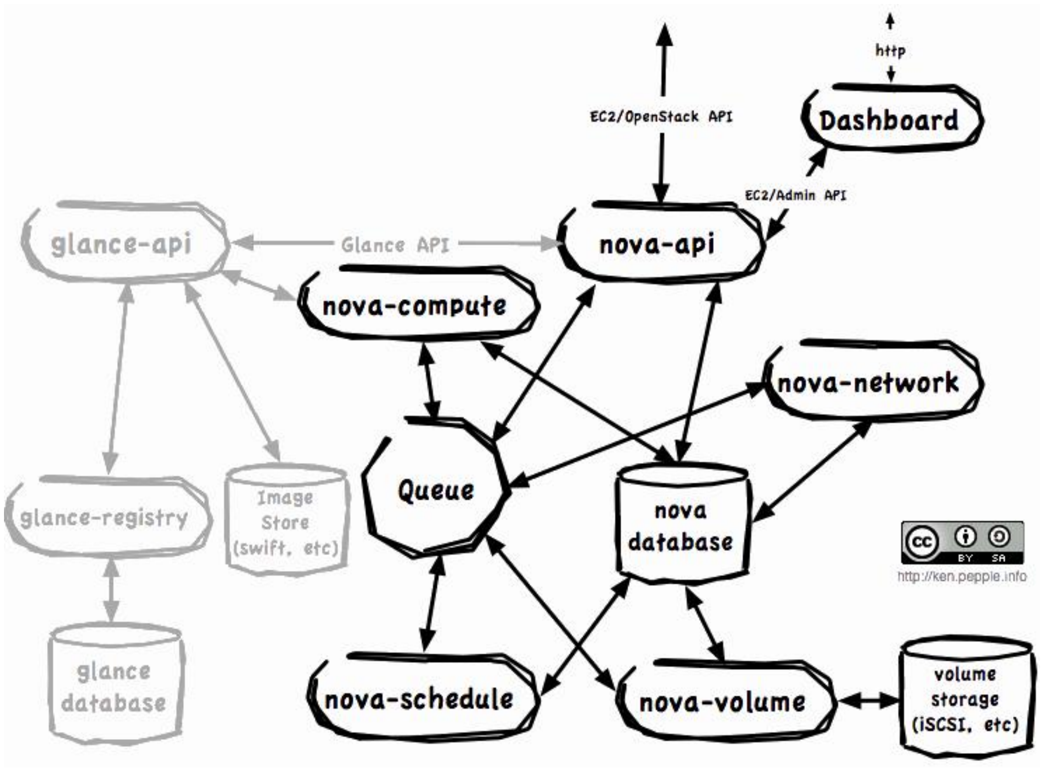


Figure 9.1: OpenStack Components (Source [Opec])

insert their requirements in a predefined form. After the user submits the request "User Requirements" are added to a predefined database schema using nova-database. The function returns UserRequirementID that uniquely identifies User Requirements in the database.

- SetInfrastructureProperties(String[] Infrastructure Properties) — Allow cloud Admin to add clouds' infrastructure properties in secure way. This is to be implemented in a form of a GUI, where a system admin insert the "Infrastructure Properties" in a predefined form. These are inserted into a predefined database schema using nova-database.
- String GetComputingNode(int UserRequirementID) — This is one of the core functions which takes as input a unique identification of the user requirements UserRequirementID. It finds the best Computing Node that can serve User Requirements.
- boolean VerifyComputingNode(int UserRequirementID, String Computing Node) — This is one of the core functions which takes a unique identification of the user requirements UserRequirementID, and the suggested Computing Node. It then verifies whether the suggested Computing Node matches user requirements as identified using the identifier UserRequirementID.

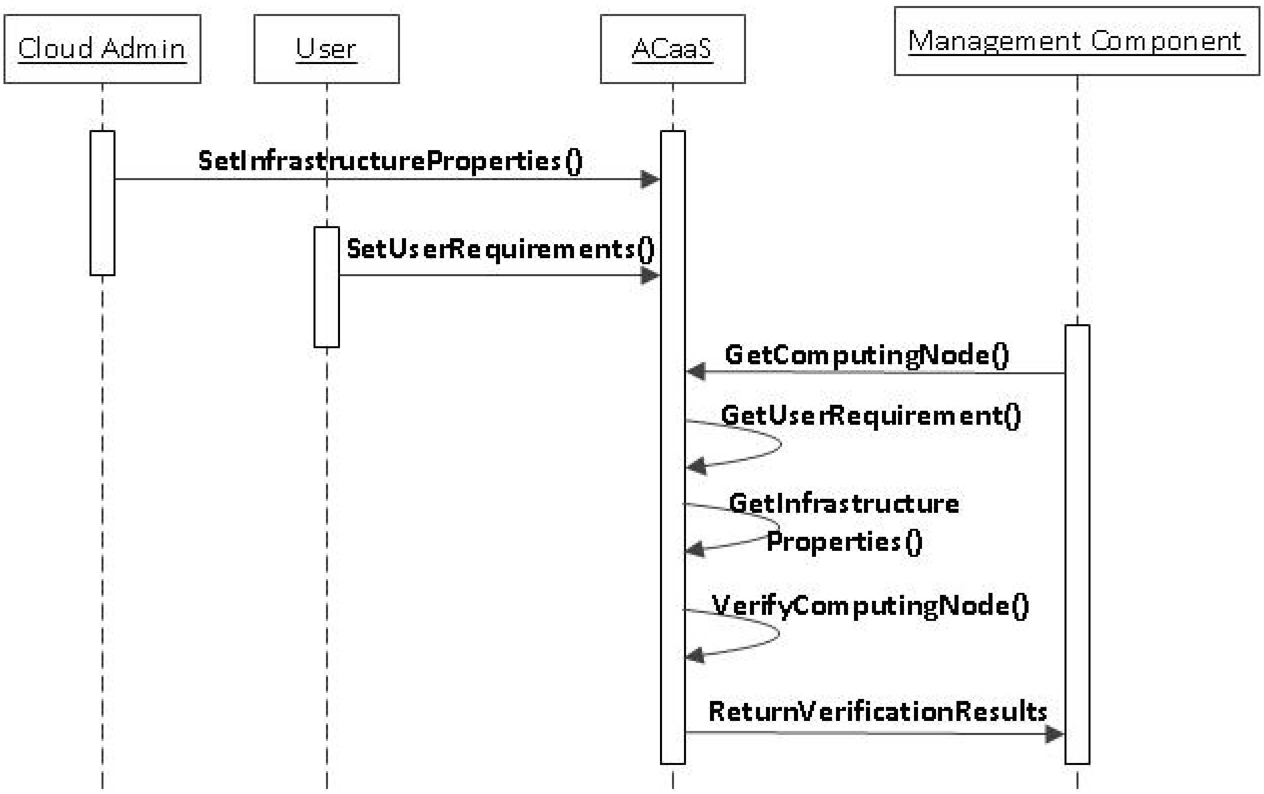


Figure 9.2: Access Control as a Service Sequence Diagram

Secure Management of Keys and VM images

9.2 TrustedObjects Manager (TOM)

Authors:
Michael Gröne, Norbert Schirmer (SRX)

9.2.1 Overview

9.2.1.1 Description

The Trusted Objects Manager (TOM) is the central management component of the trusted cloud infrastructure (cf. Deliverable D2.1.1, Chapter 12). The TOM manages the physical infrastructure including networks, services and appliances (physical platforms). Since appliances remotely enforce a subset of the overall security policy, a permanent trusted channel [GPS06], [AGS$^+$08] between the TOM and its appliances is used for client authentication, to check their software configuration using attestation, and to upload policy changes and software updates.

Finally, for each Trusted Virtual Domain (TVD) defined the TOM creates an independent TVD-specific Root-CA. SRX will contribute by enhancing the TOM to manage the Trusted-Servers within the cloud infrastructure. TOM manages TVDs and inter-TVD information flow policies, provides key-management and configures the managed TrustedServers accordingly.

As the central TVD Management Component of a TVD-based infrastructure TOM provides the user interface to define TVDs and corresponding intra-TVD and inter-TVD information flow policies.

9.2.1.2 Goals (Security, Privacy, Resilience)

- Integrity via TPM based remote attestation **Description:** The TPM is used during secure boot of the TOM to ensure its integrity. Moreover, via remote attestation the integrity can be proven to remote parties (like the TrustedServers)
- Confidentiality / integrity of key's via TPM **Description:** Private keys of the PKI infrastructure used by the TVD concepts are stored within the TPM (cf. Deliverable D2.3.1, Chapter 6).
- Web-based GUI to define Inter-TVD information flow policies and configurations **Description:** The TOM is accessed via a Web-Interface for the administrator. In Particular the allowed information flows between TVD can be defined here.
- Optional: REST based API as extension of 'standard' API like S2.
- Basic cloud API (image management, VM management)

  Description: TOM together with WP2.1 TrustedServer is planned as a replacement for OpenStack Nova. Currently TOM does not offer any cloud like API. We have to provide the API which is needed to deploy the demo applications. We have to consider if and how components of OpenStack can be reused / integrated / extended within our setting.
- Resilience / scalability via replication.

  Description: TOM provides a database of security policies (TVD based), keys, and appliances it manages, and provides the proper configurations to the appliances. TOM currently runs on a single machine and can thus only scale up to the resources of this machine and is a single point of failure. To remedy this situation replication techniques should be considered.

9.2.1.3 Required External Components

The platform shall provide a hardware Trusted Platform Module (TPM).

9.2.1.4 Relationship with Activity3

Secure management of the cloud infrastructure is a core feature of a trusted cloud, and hence needed by any application.

9.2.2 Requirements

This section gives an overview of the requirements for the TOM component.

9.2.2.1 Selected Use Cases

This use case extends /UC 20/. One can create an VM instance and select to which TVD it belongs to.

USE CASE UNIQUE ID	**/UC 460/ (Start VM instance within TVD)**
DESCRIPTION	User starts a VM instance via the TOM management interface.
ACTORS	User
PRECONDITIONS	TOM and at least one TrustedServer are up and running, TVD and VM instance are configured
POSTCONDITIONS	VM instance is deployed and started on a TrustedServer
NORMAL FLOW	1. The user selects a VM instance and a TVD and triggers the start of the VM 2. TOM selects a TrustedServer and deploys the instance on the server 3. The TrustedServer starts the VM instance

Other use cases of TOM are the definition of TVDs and of security policies. However, at this point of the project it is not yet clear what these security policies look like and how to demo them.

9.2.2.2 Demo Storyboard

Here we demonstrate the whole infrastructure and TVD concept working together.

1. Start up VM_1 in TVD_A with S3 storage attached to it (cf. /UC 460/).
2. Start up VM_2 in TVD_A with S3 storage attached to it (cf. /UC 460/).
3. Start up VM_3 in TVD_B with S3 storage attached to it (cf. /UC 460/).
4. Write a file X in VM_1 to S3 storage (cf. /UC 390/).
5. Read file X from VM_2 from S3 storage. It should be visible in plain text as TVD encryption is transparently applied and VM_1 as well as VM_2 belong to the same TVD namely TVD_A (cf. /UC 400/).
6. Try to read file X from VM_3. The file name as well as the content is only visible encrypted within VM_3 as it belongs to another TVD (cf. /UC 400/).

9.2.3 Architecture

This section describes the main components of the TOM and their respective relationships in an architectural sense.

9.2.3.1 High-level Design

The high-level design describes the architecturally significant parts of the design model, such as its decomposition into subsystems and packages. And for each significant package, its decomposition into classes and class utilities.

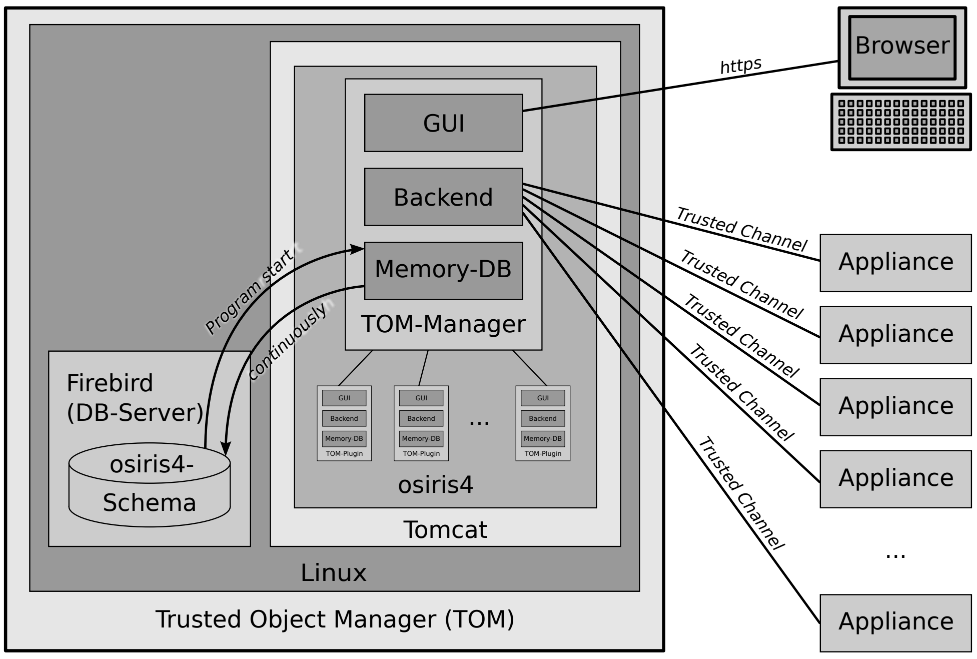


Figure 9.3: Overview of the TOM building blocks

9.2.3.1.1 Overview Internally, the TOM consists of standard PC hardware components, especially a Trusted Platform Module and two Ethernet interfaces. One for the appliances it manages and one for the management interface. The TOM runs a Linux operating system which is tied to the TPM: the hard drive is encrypted and will only decrypt with the original TPM. Moreover, the TPM is employed to generate a Public Key Infrastructure for the TVDs (cf. Deliverable D2.3.1, Chapter 6).

The TOM is running a Debian GNU Linux derivative. Inside the Linux system runs a Tomcat application server which, again, runs a web application framework called osiris4, and a Firebird database server as database backend for osiris4.

The TOM software resides as application inside osiris4: The Java part runs alongside and managed by osiris4 inside Tomcat and all persistent data is held inside the osiris4 DB.

An overview of the involved components can be found in figure 9.3.

9.2.3.2 Architecturally Significant Design Packages/Components

9.2.3.2.1 osiris4 Technically, osiris4 is used for the TOM software as an Web application framework. Many building blocks discussed below are implemented as basic functionality in osiris4. As such, osiris4 is both a manager for the TOM software (i. e. the TOM software is started by osiris4 and Web requests are handled by Tomcat, then osiris4 and then passed to the TOM software) and a library for many basic and advanced functionalities used in the TOM software. For instance, the TOM software initiates listening on the Trusted Channel, but uses osiris4 library functions for that.

Examples of library components used by the TOM software are

- declarative GUI definition,
- JSON handling,
- network access,

- binary encoding (for network messages and configuration files)

9.2.3.2.2 TOM software The TOM software is logically separated into a core component, the Manager, which handles the common infrastructure, domain objects and GUI needed by more than one security solutions (e.g. TrustedServer), and plugins that implement anything specific to a single security solution.

9.2.3.2.3 TOM Manager The TOM manager implements two network connection ”ports”: the Management Port is the interface to appliances deployed into the customers infrastructure, the Administration Port is the interface for an administrator to change the configuration of the TOM to change the configuration of connected appliances or to do other available administrative actions.

Management Port Through the Management Port the TOM controls and configures the connected appliances. Additionally, the TOM can provide the appliances with firmware updates through this channel. The TOM is always the configuration master that holds all information necessary to configure the appliances.

The Management Port runs a protocol called ”Trusted Channel” (cf. 9.3). The Trusted Channel is a TLS secured TCP/IP connection (bound to eth0 on a production TOM), that allows message based communication between two endpoints. The appliance has to demonstrate its integrity (through a TPM). Typically, the Trusted Channel is meant to be persistent. When an appliance goes offline, the semantics depend on the security solution.

The messages on the Trusted Channel are encoded in a compact binary format to facilitate frequent messages with minimum overhead. The message format can be extended for the different security solutions; an extension is provided to the TOM software by a plugin that hooks itself into the configuration data composing.

Administration Port The Administration Port is implemented as a HTTPS Web interface (bound to eth1 on a production TOM). The implementation tries to mimic desktop applications in function and appearance wherever possible. The browser runs a JavaScript application that communicates with the GUI component inside the TOM through AJAX calls (actually the ”X” part isn’t XML but JSON). Real time feedback from the server side GUI component to the browser is provided by osiris4.

Any change in the TOM configuration made through the Administration Port is instantly active and, where necessary, reflected onto the connected appliances. For this, in many cases the new valid configuration for all appliances has to be recomputed and compared to the last configuration sent to the respective appliances.

Data persistence All TOM configuration data is held in memory while the TOM software is running. On start-up **all** available configuration data is loaded, later on the config data inside the database is used write only: All configuration changes are instantly persisted into the database. The in-memory database is an optimization to be able to compute at thousands of configurations within seconds. This is reasonable since each configuration depends on many different domain objects and since the total data size easily fits into RAM.

The database may contain additional data (e.g. log data) which isn’t held in RAM.

Firmware packages, which the TOM provides to appliances, are stored on disk and held neither in the DB nor in RAM.

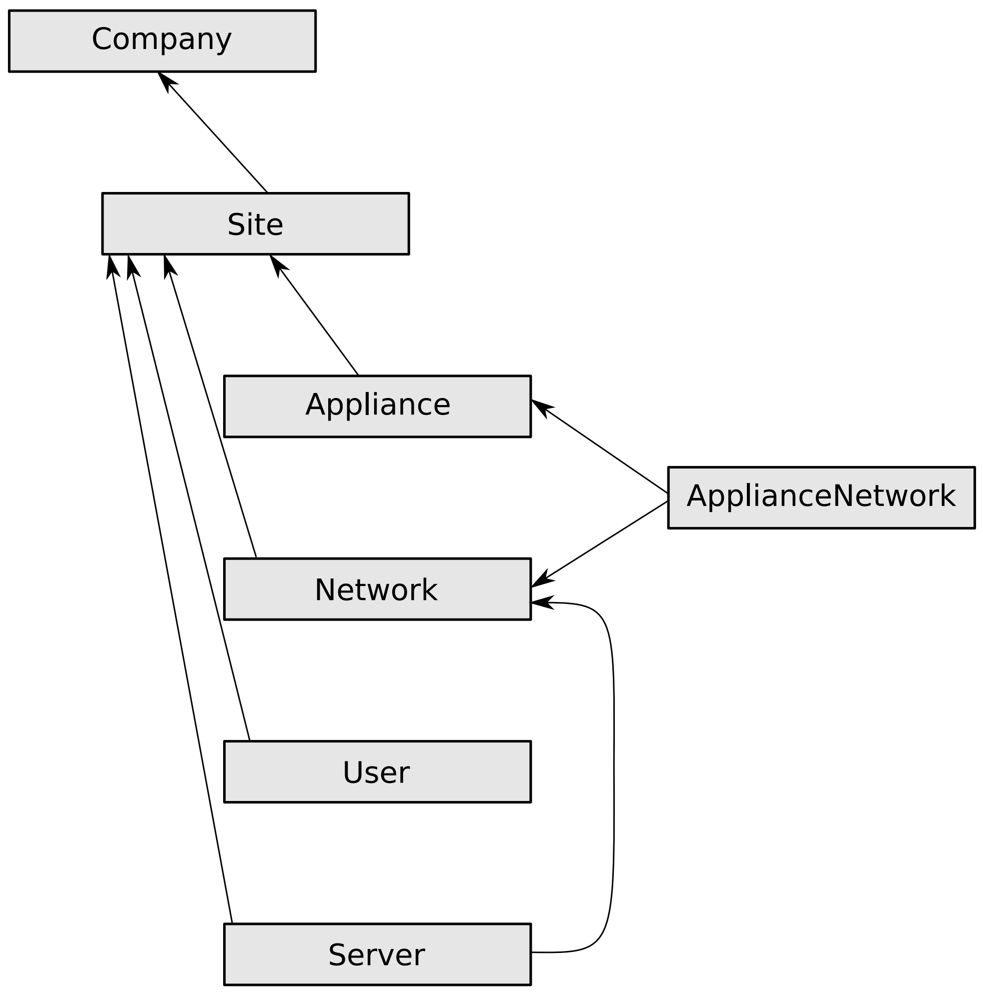


Figure 9.4: Overview of the most important TOM domain objects. Arrows indicate a 0..n–1 relationship.

Domain objects The TOM manager handles basic configuration for the following domain objects (see figure 9.4):

Organization The TOM is multi-customer capable with organization as the domain object representing the customer. Each customer's configuration data is conceptually independent from all other customer's data in the same TOM: it is linked to the organization (directly or indirectly).

Site Geographical or organizational location of one of the other domain objects. Essentially only a tag, used for grouping and filtering in the GUI and some constraint checking. Also, the link of other domain objects may go through the location.

Appliance Generic appliance connected to the TOM. Specific appliances in a security solution specialize this generic appliance.

Network An IP network (i.e. IP address, network mask et al.)

Server A server (i.e. a single IP) that resides inside a given network.

Users Administrators for the TOM (and all companies), administrators within and restricted to one organization as well as end users of a security solution.

Additionally, the following concepts reside inside the TOM manager:

Firmware Firmware software installable to the TOM or to appliances.

Deployment Group Collection of Appliances. Needed for the deployment of new firmware to the appliance.

Profiles Routing and access rules on the IP layer within one network for one class of services.

Policies Grouping of different profiles.

CertificateStore Central place for all certificates needed by the TOM.

A central organizing point inside the TOM manager is the Manager class. This class provides common functionality to the domain objects, like database connectivity, transport encoding and object registry.

Hooks for plugins The TOM software provides a plugin concept for the TOM manager. Typically, the TOM manager domain objects provide an extension interface, that can be implemented by the extending plugin. Instances, implementing such an interface register themselves with their parent object.

The trusted channel is implicitly extended to handle the configuration of the extension. Additionally, a plugin may decide to send any kind of plugin specific messages through the trusted channel.

9.2.3.3 Sequence Diagrams

TOM is shipped as a combined hardware / software appliance. To set up TOM only means to power it on (cf. Figure 9.5).

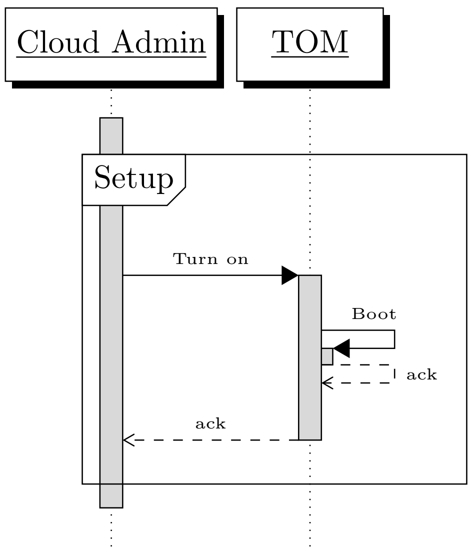


Figure 9.5: Setup of TOM.

To start a compartment in a TVD through TOM, the Project Manager has to connect to the TOM Web interface and select the compartment. TOM will choose the TrustedServer, start the compartment and get the SSH credentials for this compartment (cf. Figure 9.6).

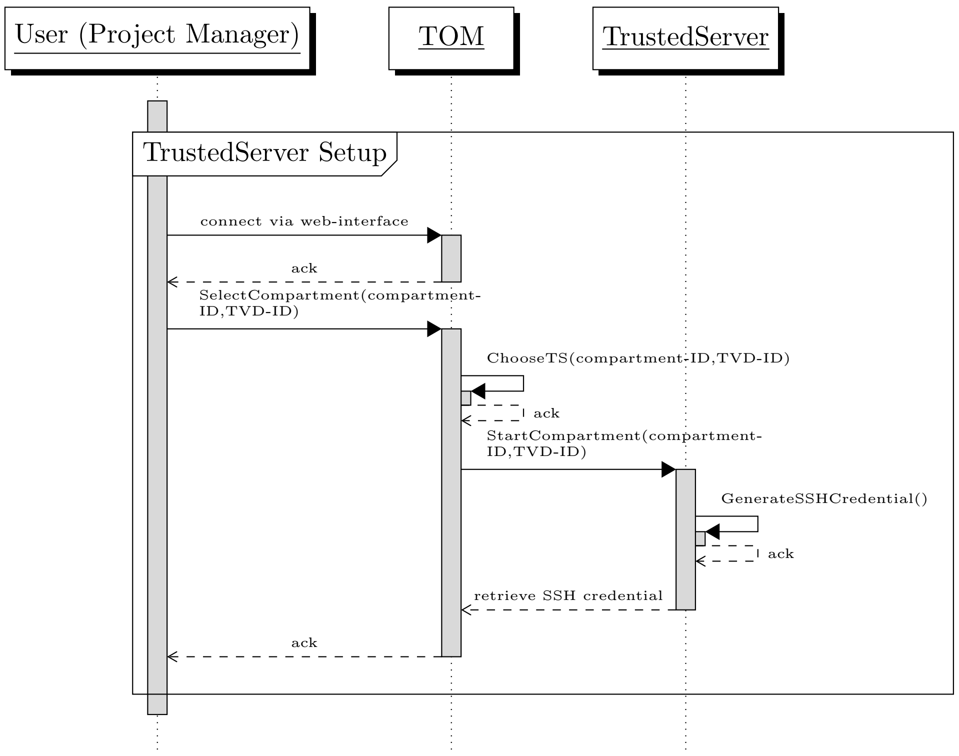

User (Project Manager)
TOM
TrustedServer
TrustedServer Setup
connect via web-interface
ack
SelectCompartment(compartment-ID,TVD-ID)
ChooseTS(compartment-ID,TVD-ID)
ack
StartCompartment(compartment-ID,TVD-ID)
GenerateSSHCredential()
ack
retrieve SSH credential
ack

Figure 9.6: Start of VM instance (compartment) in TVD by TOM.

9.2.4 API

In order to access any of the API-like functions, the GUI is needed (see High-level Design). To access any functionality of the GUI, a strong multi-factor authentication is mandatory, with factors depending on the configuration of the service. Otherwise it isn't possible to execute any of the user-level functions. Depending on the provided authentication token, access to certain functions is restricted.

9.3 Trusted Management Channel

Authors:
Michael Gröne, Norbert Schirmer (SRX)

9.3.1 Overview

This section provides an overview of the Trusted Management Channel (TMC) component. In the following we describe the overall goals and requirements, and provide an analysis of its security.

9.3.1.1 Description

The Trusted Management Channel allows to securely connect the TOM with TrustedServers to set-up, start and stop VM instances, and to load configuration and policies. It also could be used to interconnect TOMs. The Trusted Management Channel is part of the overall security concept of of Trusted Infrastructures (also referred to as TrustedInfrastructure (Sirrix)) (cf. deliverable D2.1.1, chapter 12) and all Sirrix components and products based on the TURAYA ™SecurityKernel.

9.3.1.2 Goals (Security, Privacy, Resilience)

The goal to be reached by the Trusted Management Channel is a confidential, integer, and authentic channel between management console and server using TPM based remote attestation. Its primary goal is to realize a communication channel between a client and a server that provides all required communication features (cf. subsubsection 9.3.2.3) while satisfying certain security aspects (cf. subsubsection 9.3.2.4).

9.3.1.3 Required External Components

The platform the component is installed on shall provide a hardware Trusted Platform Module (TPM) which could be replaced by a Hardware Security Module (HSM) in the future.

9.3.1.4 Relationship with other Activity2 components

The Trusted Management Channel is required by other A2 components, which are TOM, TrustedServer and Confidentiality Proxy for S3.

9.3.1.5 Relationship with Activity3 (A3)

Secure management of the cloud infrastructure is a core feature of a trusted cloud, and hence needed by any application.

9.3.2 Requirements

This section includes the requirements specification for the Trusted Management Channel component. This includes the execution/security environment, the functional requirements and the security objectives/requirements/assets.

9.3.2.1 Execution Environment

This section specifies software and hardware required at least to run this component successfully.

9.3.2.1.1 Hardware The Trusted Management Channel does not rely on special hardware in general. To be able to use certain features though, a device containing a Trusted Platform Module (TPM) is required for the Trusted Management Channel Client.

9.3.2.1.2 Software The Trusted Management Channel is designed to be operating system independent. Certain client features, such as TPM support, are currently only available on Linux platforms though.

9.3.2.2 Security Environment

This section describes the security aspects of the environment in which the component is intended to be used and the manner in which it is expected to be employed.

9.3.2.2.1 Assumptions There are three assumptions to the security aspects of the environment in which the component will be used or is intended to be used.

/A 120/ Trusted Administrator
The security administrator (a management user or the project manager) of the Trusted Management Channel Server is non-malicious.

/A 130/ Correct Hardware
The underlying hardware (e.g., CPU, devices, TPM) does not contain backdoors, is non-malicious, and behaves as specified.

/A 140/ Untrusted Cloud Administrator
The Cloud Admins of the Trusted Management Channel Client may be malicious.

9.3.2.3 Functional Requirements

Component functions that are mandatory for successful completion. Please note that security-related functions are listed in the Security Objectives (cf. subsubsection 9.3.2.4).

9.3.2.3.1 Functions Overview In the following, the identified functions are discussed grouped according to the related functional packages:

- Compatibility with TLS
- Extensibility

Compatibility with TLS

FUNCTION UNIQUE ID	**/F 10/ (Compatiblity with TLS)**
DESCRIPTION	The channel should be compatible with the TLS protocol to achieve the best results with application layer firewalls and proxies. Furthermore, it is desirable to be compatible with existing TLS implementations, i.e. to prevent modifications to TLS libraries.

Extensibility

FUNCTION UNIQUE ID	**/F 20/ (Protocol Extensibility)**
DESCRIPTION	The Trusted Management Channel should not be fixed to a single protocol method but provide an extendable interface to allow several protocol methods to coexist.

9.3.2.4 Security Objectives

The security objectives address all of the security environment aspects identified. They reflect the stated intent and shall be suitable to counter all identified threats and cover all identified organizational security policies and assumptions.

9.3.2.4.1 Security Objectives for the Trusted Management Channel

/O 160/ Mutual Authentication

The Trusted Management Channel should be able to perform mutual authentication using either a key provided by a TPM (identity key) or a shared secret (token). Authentication should take place as early as possible.

/O 170/ Remote Attestation

The Trusted Management Channel shall provide the ability to perform remote attestation within the key-based authentication scheme, to allow the server to validate the integrity of the client platform.

/O 180/ Message Secrecy and Integrity

The channel must provide message secrecy and integrity against passive and active attackers.

9.3.2.5 Security Requirements

This part of the specification defines the security requirements that have to be satisfied by the component. The statements shall define the functional and assurance security requirements that the component and the supporting evidence for its evaluation need to satisfy in order to meet the security objectives.

/SR 70/ Trust Relationship Server $\rightarrow$ Client

For the key-based authentication approach, there must be some sort of trust relationship between the identity key of each client and the server.

/SR 80/ Trust Relationship Client → Server
For any authentication approach, the client must also know if the server it is communicating with is trustworthy.

/SR 90/ TPM protected key
The key material used for key-based authentication must be protected by a TPM. The trust relationship described in /SR 70/ relies on this assumption.

/SR 100/ Security of TLS
Certain security objectives of the Trusted Management Channel are delegated to TLS. Hence, the Trusted Management Channel relies on the TLS protocol to achieve its security objectives.

9.3.2.6 Security Assets

9.3.2.6.1 Primary Assets

Communication All communication after the channel has been successfully established

Authentication Secret (Token) The token used during token-based authentication

Authentication Key (Identity Key) The identity key used during key-based authentication

Version Information (PCRs) Information about the state of the client (PCRs) might leak limited information about the client hard-/software and might hence pose a privacy problem.

9.3.2.6.2 Secondary Assets

TLS session key The key used for symmetric encryption during the TLS session

9.3.2.6.3 Threats A description of threats include all threats to the assets against which specific protection within the component or its environment is required.

/T 10/ TLS Session Key Leakage
An attacker may try to obtain the TLS session key. Using the session key, the attacker can violate the integrity and confidentiality of the channel communication.

/T 20/ Authentication Replay
An attacker may try to use previously seen information to replay authentication.

/T 30/ Session Hijacking
An attacker may try to hijack the TLS session during its authentication step.

9.3.3 Analysis

The analysis section collects analysis results, discusses relevant design alternatives, and explains the design decisions.

9.3.3.1 Security Requirements

9.3.3.1.1 Ensuring /SR 70/ and /SR 90/ To provide the /SR 70/ and /SR 90/, the trust relationship between the server and the identity key of the client is established by a trusted administrator. Each device containing an identity key is labeled with a serial number that uniquely identifies the identity key through the use of a cryptographic hash function. The production process ensures that the serial number identifies the identity key of the device and that the referred identity key is stored within the TPM. The administrator provides the serial number to the server (using a management interface) prior to any communication. With this step, the administrator ensures for the server that the specified serial number belongs to a TPM protected identity key.

9.3.3.1.2 Ensuring /SR 80/ In regular TLS, the client usually has one or more certificates issued by a certificate authority (CA) that are trusted to sign either the server certificate directly, or through a chain, in order to allow the client to verify the server's authenticity. In Trusted Management Channel, we use a more strict client verification that can however be used to achieve the same behavior as in regular TLS.

Usually, the assumption about a Public-Key-Infrastructure (PKI) is that all (sub) certificate authorities in the PKI are trusted. The Trusted Management Channel relaxes this requirement by allowing only one or more subtrees of a PKI to be considered as trusted.

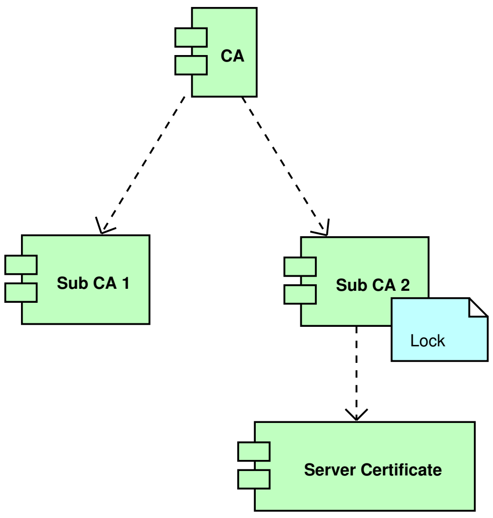


Figure 9.7: Example PKI

Assuming we have a PKI such as in Figure 9.7 and the client has to validate the server certificate, then in regular TLS, the server certificate could be signed by any of the three authorities. In our example, we would like to trust only Sub CA 2 and everything below it (even other CAs that could be below Sub CA 2, unlike in our example). We call this certificate a *lock* certificate. During the TLS handshake, the server does not only send its server certificate but also a certificate chain up to the CA of the PKI (this is part of regular TLS). In our verification, we must now ensure that the trust chain from the CA down to the server certificate contains a lock certificate. With this additional check, no certificates signed by CA or Sub CA 1 are accepted anymore (although regular TLS would do this). Note that the CA itself can be a lock certificate (in this case, the system behaves like regular TLS verification) and also the server certificate can be a lock (in this case, only exactly this server certificate is accepted).

To allow the Trusted Management Channel library to perform the additional verification, the caller must supply one or more chains, each starting with the lock certificate and ending in a CA certificate. See also 9.3.4.6.1.1 of the low-level design API for information how exactly the chains are supplied.

9.3.3.2 Functional Requirements

9.3.3.2.1 TLS Compatibility To meet the requirement /F 10/, the Trusted Management Channel uses TLS as a base for the communication channel. However, the additional security objectives, e.g. /O 170/ require further information to be transferred between the client and the server. For this purpose, the client certificate can be used, as TLS and the X.509 standard allow arbitrary extensions to be added to certificates for exactly this purpose. Furthermore, TLS provides already cryptographically secure nonces from both parties which can be used whenever such nonces are required.

Another problem is that TLS with client authentication (i.e. client certificate) requires a client key that can be used to generate the signature required by the Certificate Verify message at the end of the TLS handshake. This message consists of a non-standard signature scheme that is not compliant with PKCS1-SHA1 (the signature is PKCS1 with MD5+SHA1 concatenated instead). The only key type that is capable of performing such a signature is a `TPM_SS_RSASSAPKCS1v15_DER` key.

The security objective 'remote attestation' though requires an *attestation identity key* (AIK) to perform secure attestation actions. These two key constraints are incompatible.

Result The Trusted Management Channel design with key-based authentication will always require two different keys.

Solution We may assume that the identity key is authenticated to the server by its serial number. Attestation identity keys cannot only be used for remote attestation but also for *certification* of other keys. Using this technique, the client can authenticate the second key (called *TLS key*) using the identity key. The required for certification must be shipped within the client certificate itself like any other authentication data.

9.3.3.2.2 Extensibility The Trusted Management Channel must provide ways to support multiple authentication modes according to /F 20/. For this purpose, the client certificate must indicate what kind of authentication mode is to be used. The server must act accordingly during client certificate validation.

Result for Design The design must support multiple *handlers* for different client certificate types that represent different authentication modes.

9.3.3.3 Security Objectives

9.3.3.3.1 Mutual Authentication /O 160/)

Key-Based Authentication In this authentication mode, the client has an *attestation identity key (AIK)* which is an identity keypair $K = (pK, sK)$. The secret key sK resides within a TPM (/SR 90/) and can be used for remote attestation and key certification. The public key pK is well known. Furthermore, the client has a *TLS keypair* $TK = (pTK, sTK)$ as described in paragraph 9.3.3.2.1. The secret key sTK also resides in the TPM. This keypair can be generated on-the-fly or kept statically and is not known to the server beforehand.

Prior to any communication between the server and the client, the serial number (or fingerprint) S of the public key pK must be authorized within the server (e.g. through a system administrator) as described in paragraph 9.3.3.1.1.

Once this step is complete, the serial number S and hence the keypair (pK, sK) is considered trustworthy by the server. Furthermore, the client has a list of certificate chains that are trusted for signing server certificates as described in paragraph 9.3.3.1.2.

With these prerequisites, mutual authentication can be achieved with standard TLS using client and server certificates.

In the following, the different steps of the TLS handshake are explained in detail.

(1) Client/Server Hello To establish a connection between the client and the server, the client first issues a *TLS Client Hello* to the server. The server responds with a *TLS Server Hello*. By the TLS specification, both messages contain a certain amount of random data (28 byte randomness + 4 byte timestamp). By $nonce_S$ we denote certain amount of bits (at least 160) of the randomness included in the *server hello* message.

(2) Server Authentication The server sends its certificate to the client, including the server public key pK_S. This certificate is signed by a trusted authority and can hence be validated by the client. Furthermore, the server sends the *ServerKeyExchange* message, which in general contains all necessary parameters for a key exchange. In our case, we perform the key exchange using RSA and this message contains the necessary RSA parameters (e.g. modulus, exponent). This message is signed by the server.

(3) Client Authentication The server now requests a certificate from the client for authentication. At this point, the client obtains the secondary keypair $TK = (pTK, sTK)$ residing in the TPM and constructs an X509 certificate, including the information as described in table 9.8. The certificate is *self-signed*.

(4) Handshake Completion In order to complete the handshake, the client must now encrypt a *pre-master secret* using pK_S and send it to the server. After the handshake is completed, both parties can use this to derive the *master secret*. Finally, the client sends the *CertificateVerify* message. This message contains a signature over all previous handshake messages and is performed using sTK. The client performs this message both to prove that it possesses the secret key that matches the public key shown in the client certificate and to ensure the integrity of all previous client messages.

By $ci(pK, pTK)$ we denote the output of a TPM_CertifyKey call to authenticate the key pTK through the known TPM key pK. Without this call, there would be no binding between these two keys and no assurance that the key pTK is really in possession of the client.

Inclusion of remote attestation specific data (Trusted/Actual Version Information) is optional. The server may enforce the presence of this data for certain types of clients (e.g. appliances). For more information about these fields, see 'remote attestation'.

The presence of the $nonce_S$ within the client certificate makes the whole certificate *unique* for this session and prevents replay attacks on the certificate (/T 20/).

Important: The primary keypair $K = (pK, sK)$ **must** be an attestation identity key (AIK). If a `TPM_SS_RSASSAPKCS1v15_SHA1` signing key would be in use, the attacker could fake the

Information	Purpose
Public Key pTK	Public key of X509 Certificate S (**signing**)
Public Key Info/Signature $ci(pK, pTK)$	Certification for pTK by pK (**key authentication**)
Public Key pK	Allow the server to derive the serial number S (**identity**)
Trusted Version Information	Allow the server to validate our configuration during remote attestation (PCRs)
Actual Version Information	Provide the server with information about current configuration (PCRs) (**remote attestation**).
$nonce_S$	The nonce that is used during the handshake and for quote (informative, additional **replay protection**)
sign(sTK, Hash(ClientCertificate))	Prove that the certificate originates from the client and was not altered (part of X509, **integrity**)

Figure 9.8: Data to be included in the client certificate for key-based TLS

output of TPM_Quote using this key, as there is no way to distinguish the output of TPM_Sign and TPM_Quote with this key type.

Token-Based Authentication In this authentication mode, the client knows some *token* and in order to succeed in authentication, the client has to prove the server that it knows a valid token.

Because the client has nothing except the token to prove its identity, we must not only ensure that the token is kept secret during transmission, but also that the token submission is bound to the client and the session (see /T 20/ and /T 30/).

The respective steps in the TLS handshake are identical to the steps in the *key-based authentication*, except for the client authentication step:

(3) Client Authentication The client generates an asymmetric key $K = (pTK, sTK)$ and constructs a client certificate, including the information as described in table 9.9. The certificate is again **self-signed**.

After seeing the client certificate, the server knows what token the client is referring to by decrypting $Enc(pK_S, token)$. The correctness of the HMAC value can then be easily verified by the server. The presence of the $nonce_S$ in the HMAC prevents replay attacks (/T 20/).

The handshake is then completed with step 4 which is identical again to the *key-based authentication*.

9.3.3.3.2 Remote Attestation /O 170/ Remote attestation can only be supported in the key-based authentication scheme. To perform remote attestation, the server needs two additional datasets that must be included in the client certificate:

Trusted Version Information The server needs to know, what configurations are allowed. This information could be hardcoded within the server but this approach is inflexible and does not scale. A better approach is to let the client possess a proof that a certain configuration is trusted. This can be realized by using a *PCR Certificate Authority (PCR CA)* to sign trusted

Information	**Purpose**
Public Key pTK	Allow the server to validate the client signature later (**integrity**)
Encrypted Token $Enc(PK_S, token)$	Let the server know which token we have
HMAC(token, $nonce_S$ || pTK)	Provide the server with evidence about the client *currently* knowing the token and bind pTK to the token (**identity/integrity**)
$nonce_S$	The nonce that is used during the handshake and for HMAC (informative, additional **replay protection**)
sign(sTK, Hash(ClientCertificate))	Prove that the certificate originates from the client and was not altered (**integrity**)

Figure 9.9: Data to be included in the client certificate for token-based TLS

configurations. The server can verify that the information is valid only by knowing the PCR CA.

Actual Version Information The server needs to know the actual configuration of the client. This can be achieved using TPM_Quote(sK, $nonce_S$, PCRSel) on the client. This will yield non-replayable evidence about the state of the clients' PCR registers (simplified sign(sK, PCRs || $nonce_S$)). The actual PCR selection to use is fixed by the trusted version information which the client possesses. To verify the signature, we also need to include the PCRs in question though. As described in paragraph 9.3.2.6.1, this information might violate the privacy of the client under certain circumstances. The information is hence to be encrypted with the server public key pk_S.

9.3.3.3.3 Message Secrecy and Integrity /O 180/ As described in paragraph 9.3.3.2.1, the Trusted Management Channel uses TLS to establish the secure channel. The desired secrecy and integrity properties are therefor provided by the TLS layer itself, given that TLS is secure (/SR 100/). The security of TLS also depends on the ciphersuite used. For the purposes of Trusted Management Channel, both the server and the client shall be using **AES128-SHA** as ciphersuite.

This mode uses RSA to exchange a pre-master secret which is then used by server and client to derive a 128 bit master secret. The master secret is used with the symmetric AES algorithm, providing sufficient **secrecy**. To provide **integrity**, TLS uses a *Message Authentication Code (MAC)* with a given hash algorithm, in this case SHA1.

9.3.3.4 Demo Storyboard

There is no direct interaction of an user (User, Cloud Admin, Developer) with the TrustedChannel. Demo is included in demo of TOM (cf. 9.2, section 9.2.2.2), TrustedServer (cf. 7.5, section 7.5.2.7) and Confidentiality proxy for S3 (cf. 8.4, section 8.4.2.7) components.

9.3.4 Architecture

9.3.4.1 High-level Design

The following section gives a brief overview of the high-level design.

9.3.4.2 Components

Figure 9.10 provides a simplified version of the component relations within the Trusted Management Channel. We will now briefly describe the purpose of each component/subsystem.

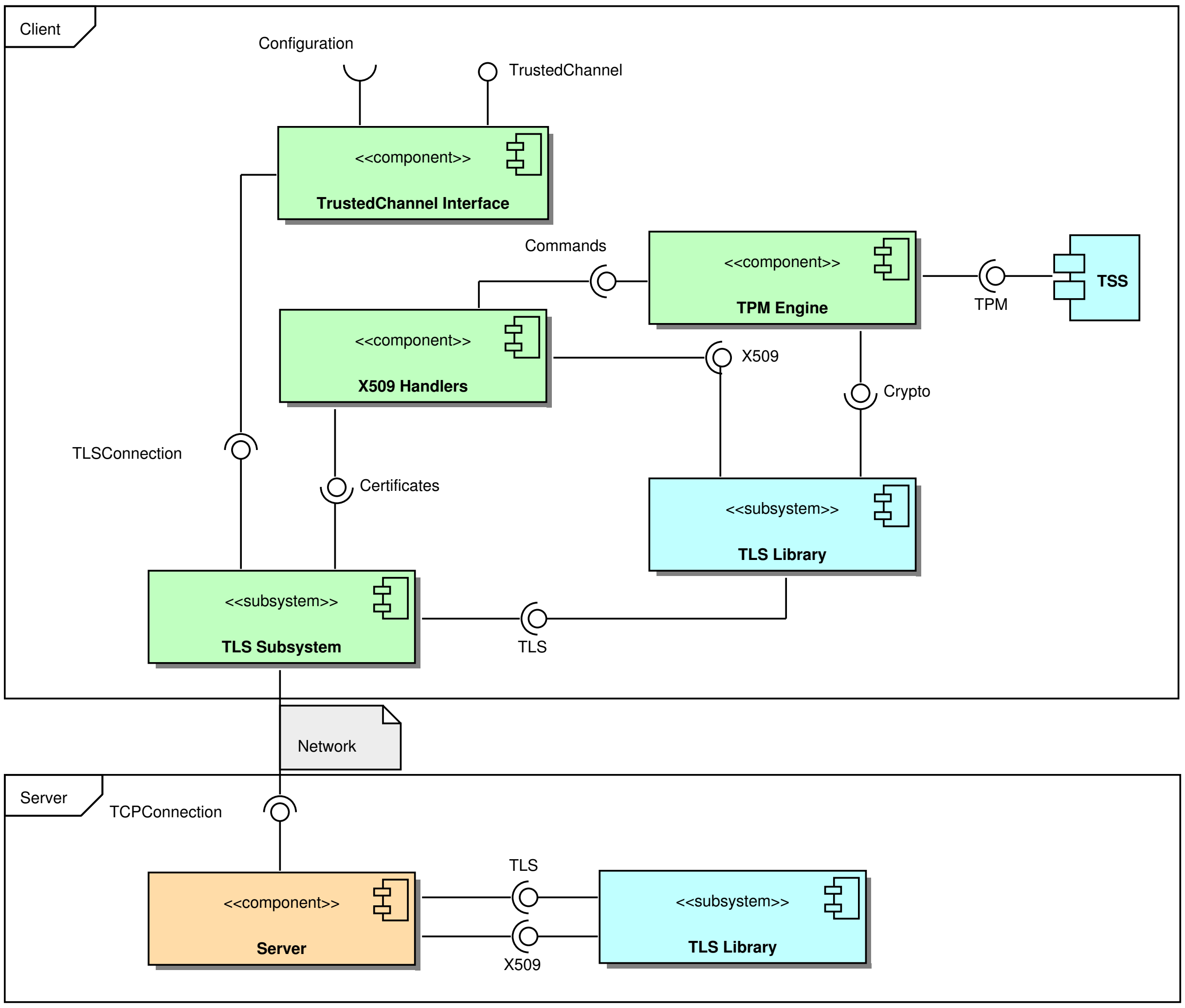


Figure 9.10: Component overview

Trusted Software Stack (TSS)

The *Trusted Software Stack (TSS)* is an external library that provides an interface to the TPM itself. This component is required for all operations that require a TPM.

TLS Library

This *TLS library* is another external library providing several TLS/Crypto related functions. This includes amongst others X509 related functionality, cryptographic algorithms and an implementation of the TLS network protocol.

Engine

The Engine component has two main purposes. First, it serves as an *abstracted interface* between the TLS Library and the TSS by providing certain *methods* related to cryptographic purposes (*RSA* and *randomness*) in a form specific for the TLS library. Secondly, the engine provides additional *commands* that perform certain actions with the TPM and can be used directly by other components within the Trusted Management Channel Client. The engine concept allows the design to stay independent from the underlying TSS library.

X509 Handlers

The *X509 handlers* provide everything related to *X509 certificates* within the Trusted Management Channel. This especially includes the different authentication forms the Trusted Management Channel must be able to support. Adding a new authentication form can simply be achieved by implementing an additional X509 handler that provides the required authentication certificate.

TLS Subsystem

The *TLS Subsystem* provides the TLS connection by involving the *TLS library* together with certificates provided by the *X509 handlers*. This component also establishes the connection to the Trusted Management Channel Server.

TrustedChannel Interface

The TrustedChannel is the public interface available to the developer. It requires system-dependent configuration and provides the channel itself.

9.3.4.3 TPM-based Connection

With the TPM-based authentication, the channel object is first instantiated with a configuration (containing TPM-related settings such as key files or NVRAM indices). Then, the daemon may open the channel, specifying the host, port and CA certificate to use for this step. The library opens a connection to the specified server and obtains the server certificate. It then performs certain steps with the TPM engine to collect the necessary data for certificate generation and then submits a client certificate. Finally, the TLS handshake is continued and the connection is open if the handshake succeeded (cf. figure 9.11).

9.3.4.4 Token-based Connection

The Token-based connection is similar to the TPM-based connection. Instead of TPM-related data, the configuration during instantiation contains the token to be used. When the daemon

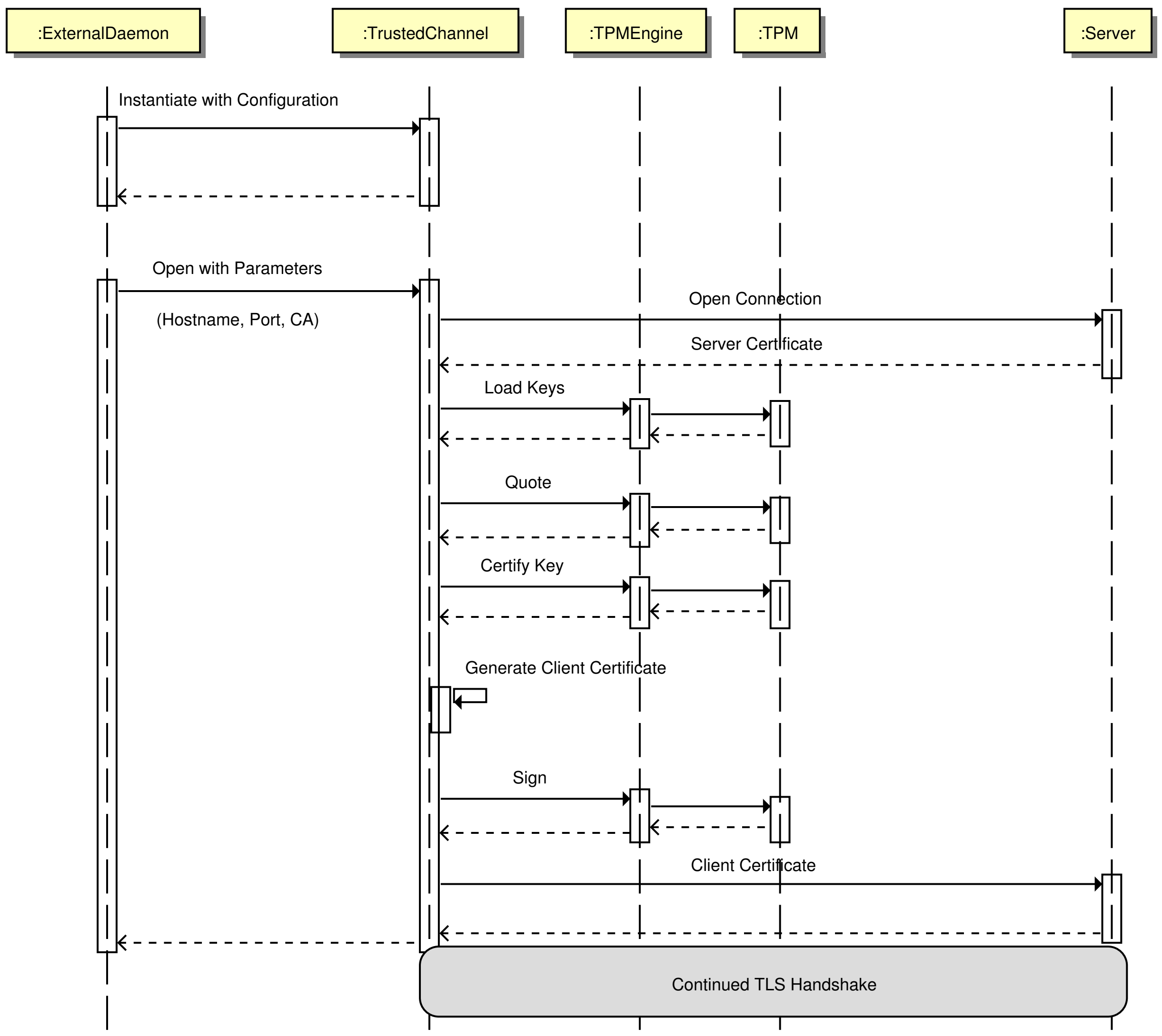


Figure 9.11: TPM-based connection

opens the channel, the library performs different steps (encrypt/HMAC token) to produce the client certificate and continue the handshake (cf. figure 9.12).

9.3.4.5 Sequence Diagrams

The TMC is an internal component and thus has no direct interaction with the User. It is used for communication of TOM (server) and an appliance (client) in our case a TrustedServer. The TMC is accessed via a library at both ends. We describe mutual authentication which is used as a first step to establish an communication channel.

9.3.4.5.1 Mutual Authentication - Key-Based Authentication (Figure 9.13) The whole process of a a successful key-based authentication as described in *key-based authentication* is depicted in Figure 9.13 for an appliance that also performs remote attestation.

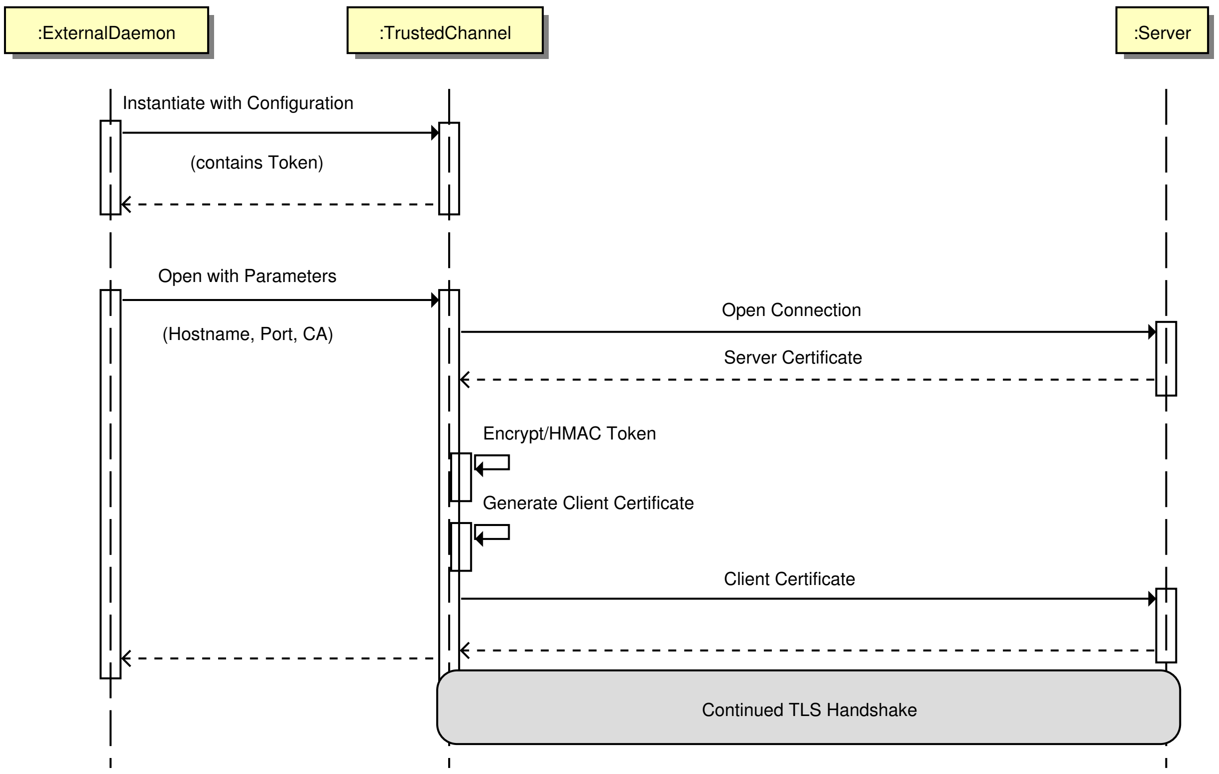


Figure 9.12: Token-based Connection

9.3.4.5.1.1 Mutual Authentication - Token-Based Authentication (Figure 9.14) The whole process of a a successful token-based authentication as described in *token-based authentication* is depicted in Figure 9.14.

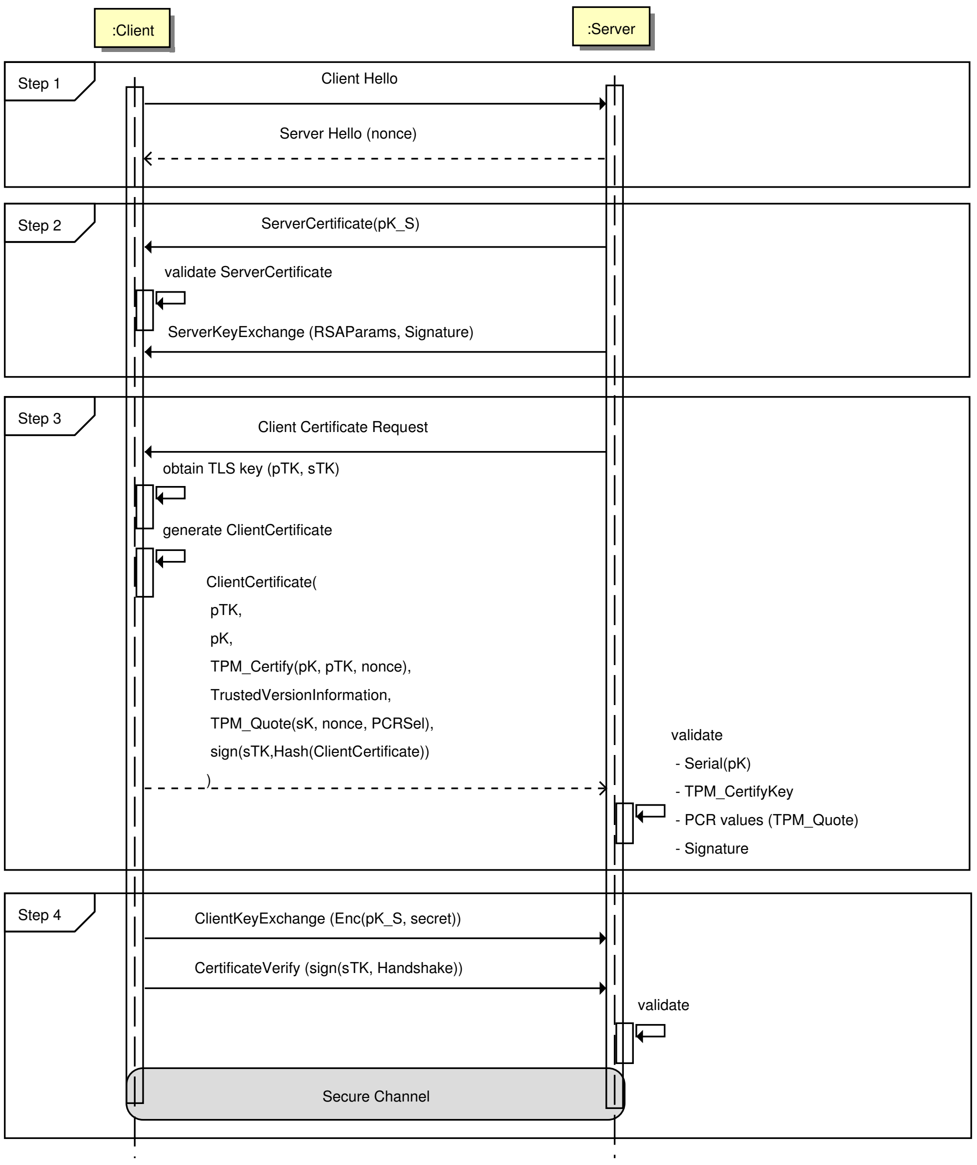
9.3.4.6 Low-Level Design

This chapter provides a more detailed specification of the Trusted Management Channel design with regards to the structures/certificates used as well as the Trusted Management Channel Client library design and a functional specification of the Trusted Management Channel Server. Implementation languages are C++ and Java.

9.3.4.6.1 Client Library Design In this section we give a more detailed description for the classes used in the Trusted Management Channel library. Figure 9.15 gives an overview of the classes. We will describe their purpose now in detail.

9.3.4.6.1.1 Client TLS Library For the client design, the **OpenSSL** TLS library seems to be the best choice because it is widely supported and provides the necessary *engine* interface for supporting the TPM.

Engine Requirements The engine must provide certain features in order to fulfill the requirements of this design. In terms of OpenSSL engines, the engine must provide a `RAND_METHOD` as well as parts of the `RSA_METHOD` (encryption with the private key, i.e. signing). Furthermore, it must provide a mechanism to load the key from file and from NVRAM and provide it as an `EVP_PKEY` object to the OpenSSL API. Finally, the engine should supply control commands to perform *TPM_Quote()* and *TPM_CertifyKey()* with the key(s) provided. Later versions



:Client
:Server
Step 1
Client Hello
Server Hello (nonce)
Step 2
ServerCertificate(pK_S)
validate ServerCertificate
ServerKeyExchange (RSAParams, Signature)
Step 3
Client Certificate Request
obtain TLS key (pTK, sTK)
generate ClientCertificate
ClientCertificate(
pTK,
pK,
TPM_Certify(pK, pTK, nonce),
TrustedVersionInformation,
TPM_Quote(sK, nonce, PCRSel),
sign(sTK,Hash(ClientCertificate))
)
validate
- Serial(pK)
- TPM_CertifyKey
- PCR values (TPM_Quote)
- Signature
Step 4
ClientKeyExchange (Enc(pK_S, secret))
CertificateVerify (sign(sTK, Handshake))
validate
Secure Channel

Figure 9.13: Sequence diagram for a successful key-based authentication

of both engines furthermore introduce a *ENGINE_VERSION* control command which returns a hard-coded version number to check if the engine is compatible with the Trusted Management Channel.

TrustedChannel The *TrustedChannel* is an abstract base class that is derived by classes implementing the Trusted Channel, e.g. *BinaryTrustedChannel* which implements the Trusted Channel for binary transmission. Every implementation of this class implements a constructor which takes a *ChannelConfig* containing the configuration parameters for the channel. Furthermore, they provide an *open()* method, expecting server hostname/port and one or more trust chains for creating a connection. Trust chains are supplied as single files, each file containing one or more certificates in PEM format. The first certificate in each file is considered to be the lock certificate, the last certificate must be the CA (see paragraph 9.3.3.1.2 for a documentation on trust chain semantics). Finally a *close()* method is provided for closing the connection again.

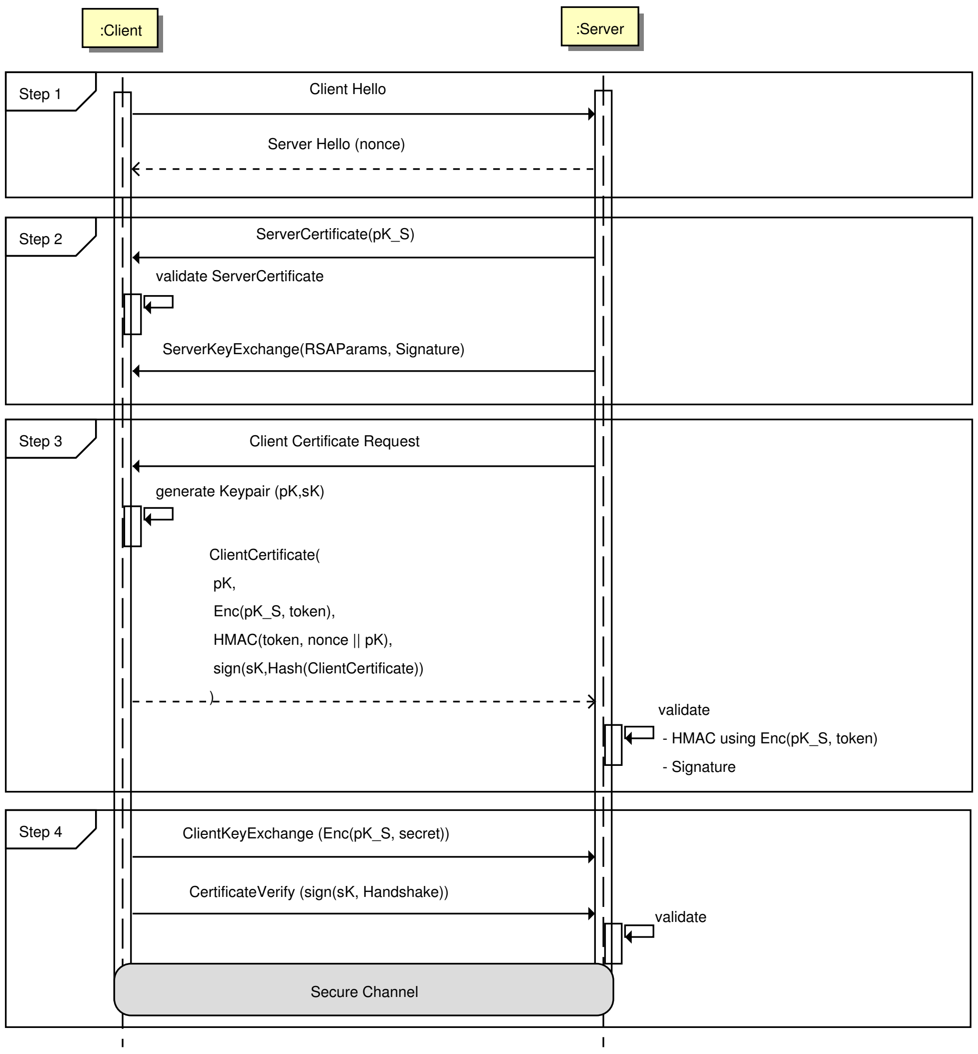

:Client
:Server
Step 1
Client Hello
Server Hello (nonce)
Step 2
ServerCertificate(pK_S)
validate ServerCertificate
ServerKeyExchange(RSAParams, Signature)
Step 3
Client Certificate Request
generate Keypair (pK,sK)
ClientCertificate(pK, Enc(pK_S, token), HMAC(token, nonce || pK), sign(sK,Hash(ClientCertificate)))
validate - HMAC using Enc(pK_S, token) - Signature
Step 4
ClientKeyExchange (Enc(pK_S, secret))
CertificateVerify (sign(sK, Handshake))
validate
Secure Channel

Figure 9.14: Sequence diagram for a successful token-based authentication

Figure 9.16 also gives an overview of the related classes.

During the *open()* call, several exceptions may occur due to external circumstances:

Exception	**Meaning**	**Possible Reason(s)**
CException	The connection failed on the socket level	The host/port is unreachable or closed
SSLConnectionException	The connection failed on the handshake level	The server rejected the client (e.g. certificate, other policy)
SSLSecurityViolationException	The connection succeeded but the verification of the server failed	Invalid server certificate

BinaryTrustedChannel The *BinaryTrustedChannel* implements the *TrustedChannel* for binary transmission and implements a binary stream.

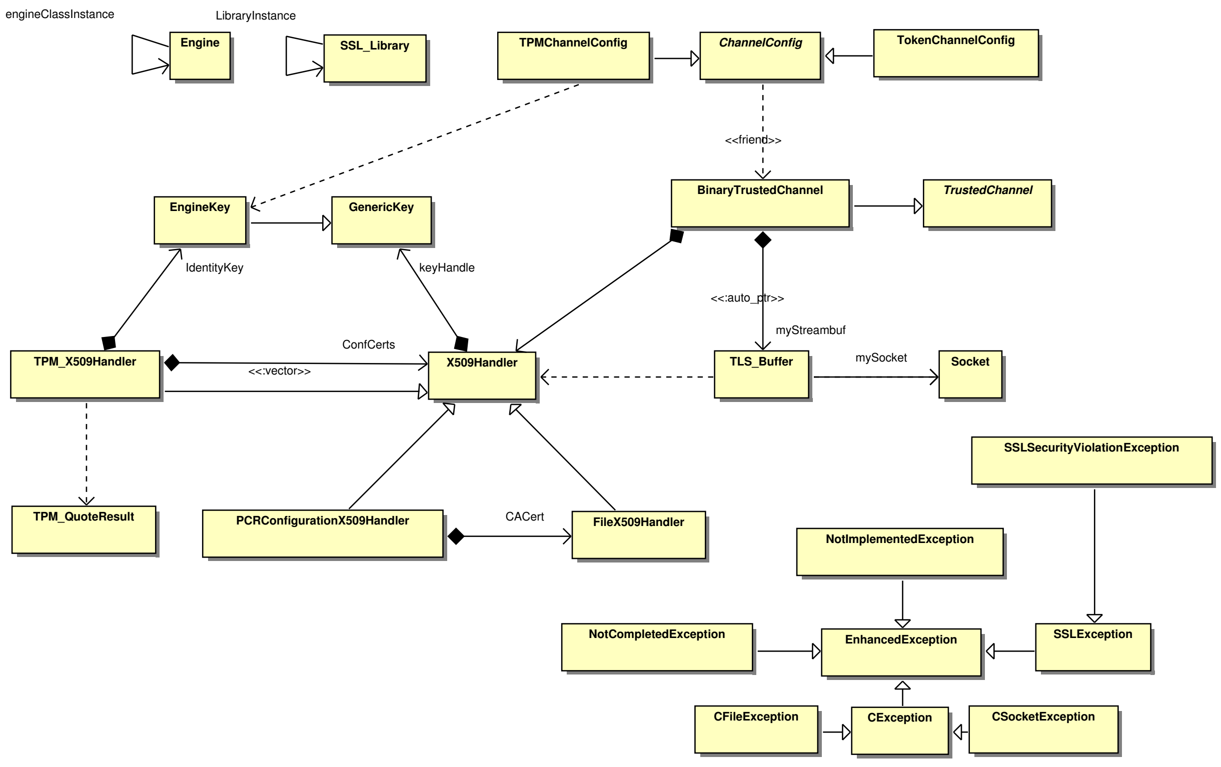


Figure 9.15: Class diagram of the Trusted Management Channel library design

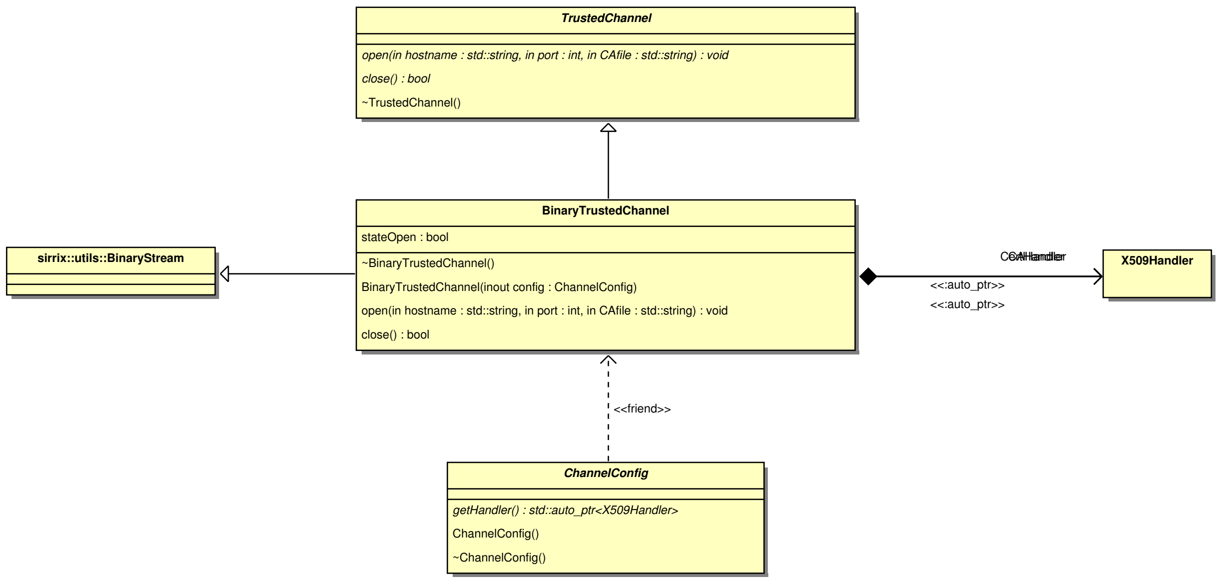

TrustedChannel
open(in hostname : std::string, in port : int, in CAfile : std::string) : void
close() : bool
~TrustedChannel()
BinaryTrustedChannel
stateOpen : bool
~BinaryTrustedChannel()
BinaryTrustedChannel(inout config : ChannelConfig)
open(in hostname : std::string, in port : int, in CAfile : std::string) : void
close() : bool
sirrix::utils::BinaryStream
CAHandler
<<:auto_ptr>>
<<:auto_ptr>>
X509Handler
<<friend>>
ChannelConfig
getHandler() : std::auto_ptr<X509Handler>
ChannelConfig()
~ChannelConfig()

Figure 9.16: Classes related to the TrustedChannel

9.3.4.6.1.2 ChannelConfig The *ChannelConfig* is an abstract class that represents a configuration for the *TrustedChannel*. Every implementation of this class represents a possible configuration mode. There are currently two configurations.

TPMChannelConfig The *TPMChannelConfig* takes two keys (one key that should be used for Identity/Remote Attestation and another key to perform the TLS handshake) and one or more filenames referring to either configuration certificates (PCR values) or intermediate CA certificates used as trusted chain for the configuration certificates.

The keys can either be in the form of files (sealed keyblobs) or indices referring to key slots in the NVRAM of the TPM.

When passing this configuration object to the *TrustedChannel*, the channel will run the key-based authentication protocol.

TokenChannelConfig The *TPMChannelConfig* only takes a token (a string). When passing this configuration object to the *TrustedChannel*, the channel will run the token-based authentication protocol.

9.3.5 API

The Trusted Management Channel is an internal protocol which mainly consists of the TLS protocol API and internal extensions. It is integrated into TOM component and has one public interface.

- TLS plus TPM-based extensions
- Management protocol for RPC.

9.3.5.1 Public Interface

The Trusted Management Channel library can be used by instantiating a subclass of *TrustedChannel*, for example *BinaryTrustedChannel* for binary transmission.

Figures 9.17 and 9.18 describe the public interface:

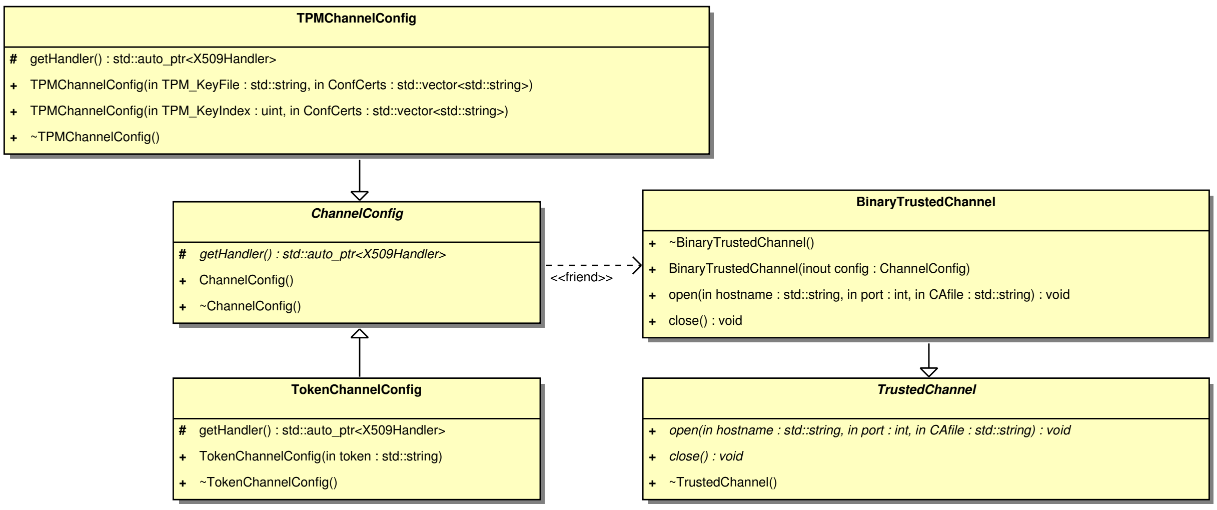

TPMChannelConfig
# getHandler() : std::auto_ptr<X509Handler>
+ TPMChannelConfig(in TPM_KeyFile : std::string, in ConfCerts : std::vector<std::string>)
+ TPMChannelConfig(in TPM_KeyIndex : uint, in ConfCerts : std::vector<std::string>)
+ ~TPMChannelConfig()
ChannelConfig
# getHandler() : std::auto_ptr<X509Handler>
+ ChannelConfig()
+ ~ChannelConfig()
<<friend>>
BinaryTrustedChannel
+ ~BinaryTrustedChannel()
+ BinaryTrustedChannel(inout config : ChannelConfig)
+ open(in hostname : std::string, in port : int, in CAfile : std::string) : void
+ close() : void
TokenChannelConfig
# getHandler() : std::auto_ptr<X509Handler>
+ TokenChannelConfig(in token : std::string)
+ ~TokenChannelConfig()
TrustedChannel
+ open(in hostname : std::string, in port : int, in CAfile : std::string) : void
+ close() : void
+ ~TrustedChannel()

Figure 9.17: Public interface of the *TrustedChannel*

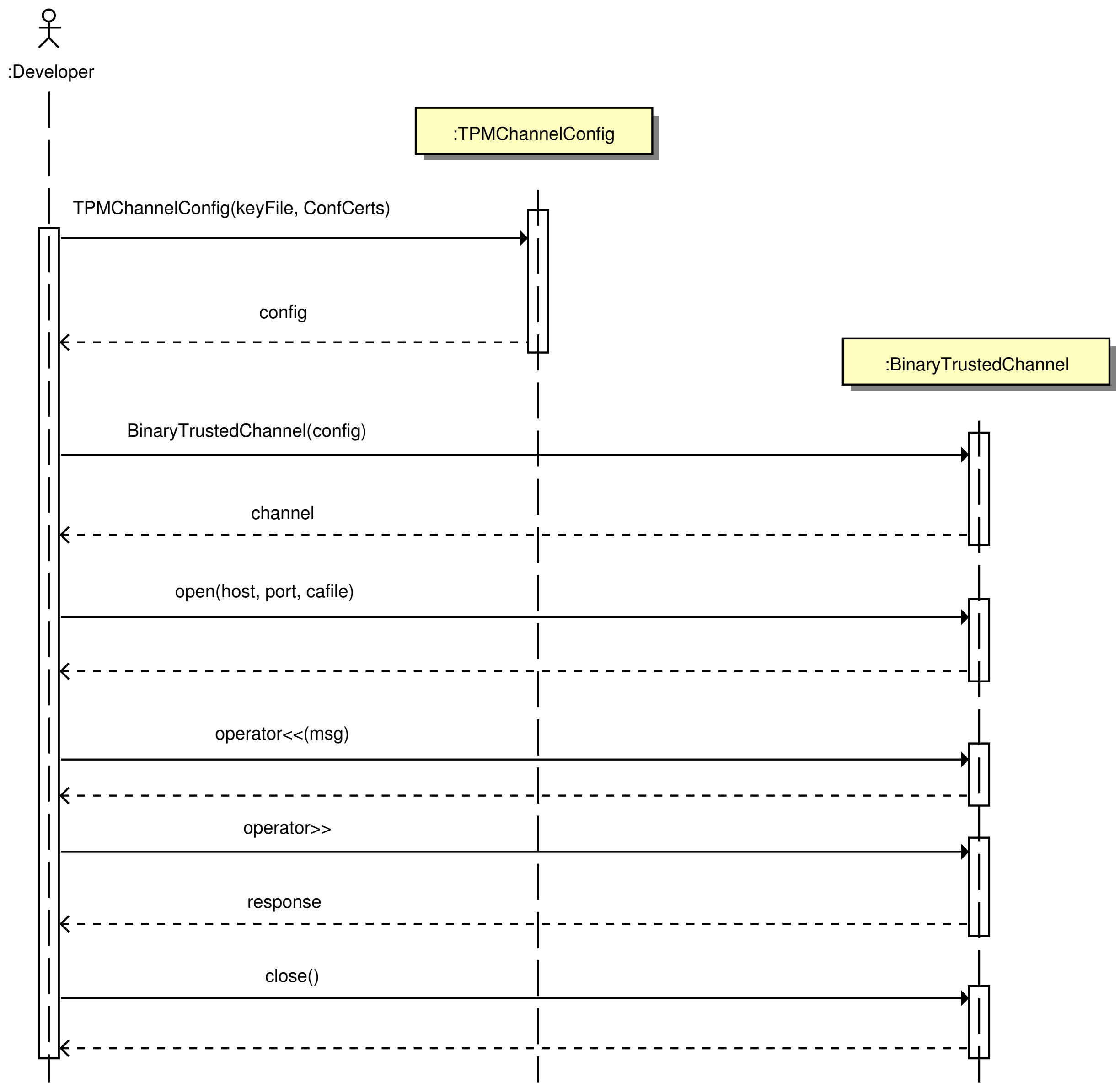


Figure 9.18: Sequence diagram for TrustedChannel usage

9.3.5.1.1 Example Usage of the Public Interface Figure 9.19 shows a small program that opens a channel using the TPM and sends ”Hello World!” to the server. It also shows the main steps to use the *TrustedChannel*:

- Create a configuration (based on what type of channel you need)
- Create the channel with the given configuration
- Open the channel with the specified connection parameters
- Send/receive data using the streaming operators (flush() if required)
- Close the channel

```
#include <Sirrix/TrustedChannel/BinaryTrustedChannel.hxx>
#include <Sirrix/TrustedChannel/TPMChannelConfig.hxx>

using namespace sirrix;
using namespace tc;

int main() {
   /* Get us a vector of PCR Certificates (PEM Format) */
   std::vector<std::string> PCRCerts;

   PCRCerts.push_back("./myPCRCert1.pem");
   PCRCerts.push_back("./myPCRCert2.pem");
   PCRCerts.push_back("./myPCRCert3.pem");

   /* Create the configuration object */
   TPMChannelConfig config("./myIdentityKeyFile", "./myTLSKeyFile",
      PCRCerts);

   /* Create the channel */
   BinaryTrustedChannel tc(config);

   /* Open the channel */
   try {
      tc.open("localhost", 4433, "./myCACert.pem");
   } catch (CException &cexc) {
      std::cerr << "Connection failed on socket level" << std::endl;
      return 1;
   } catch (SSLConnectionException &sslcexc) {
      std::cerr << "Connection failed on handshake level" << std::endl;
      return 1;
   } catch (SSLSecurityViolationException &sslsvexc) {
      std::cerr << "Server certificate verification failed" << std::endl;
      return 1;
   }

   /* Send hello world and flush the channel (it has a 4k buffer) */
   tc << "Hello world!";
   tc.flush();

   /* Close the channel */
   tc.close();
}
```

Figure 9.19: Simple ”Hello world” with the Trusted Management Channel

Verification and Auditability

9.4 Ontology-based Reasoner to Check TVD Isolation

Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

9.4.1 Overview

The ontology-based Reasoner is a subcomponent/plugin for the Management Component that, given as input a service model, an infrastructure model and an allocation of services onto the infrastructure, makes it possible to verify whether some security properties required by the service are satisfied by the allocation. Furthermore, it may also provides hints on how to modify the allocation whenever security requirements are not met.

More specifically, the service model shall describe a TVD as a virtual network and the main property we shall verify is isolation, in part achieved at "computational" level by the hypervisor, in part achieved at network level by securing untrusted channels. We rely on ontology-based reasoning to perform analysis.

9.4.1.1 Goals (Security, Privacy, Resilience)

- Isolation (focus on network level).
  Description: The ontology-based Reasoner should be able to verify that all communications inside a service are kept isolated (focus on confidentiality and integrity) from other TVDs deployed on the same cloud infrastructure. Otherwise, the tool should be able to provide a new configuration, e.g. by requiring to establish some secure channels, that meets the requirements. We also plan to work on dynamic (re)configuration.
  Techniques/research problems: Ontology-based reasoning. Define a model for services (TVDs), infrastructure, allocation, and a consistent attacker model.
  Assumptions: N/A.

- Description of security services (optional)
  Description: Optionally, in our service model we would like to capture some of the security services provided by TClouds, e.g. a resilient storage. This means that the service model shall describe at higher level something like "service A requires the resilient(*params*) storage service B", this description is translated in a precise allocation (e.g. service B is mapped onto (sub)services $B_1, \cdots, B_{2f}$ with a description of the interactions among them) and finally the tool perform the analysis.
  Techniques/research problems: ontology-based description of security services.
  Assumptions: The model of the security services provided by TClouds is statically included into the reasoner.

9.4.1.2 Required external components

- Management Component (if integration is needed).
  Name/description: If integration is required, we plan to plug our tool into the existing OS Management Component.

Features (security/privacy/resiliency): N/A.
Required API (provided by the external component): XML-based format for system model, e.g. the `libvirt` XML format (not really an API).

- Infrastructure capable of securing channels among, e.g., cluster nodes.
  Name/description: We expect our tool to be able to decide if a secure channel must be added (or can be removed if it is redundant). The infrastructure should be able to change the configuration. We plan to add support to `libvirt` to establish secure channels (IPsec) among Cloud Nodes.
  Features (security/privacy/resiliency): N/A.
  Required API (provided by the external component): `libvirt`.

9.4.1.3 Relationship with Activity3

Actors from A3 act as end users of the cloud and therefore can access the Log Service. For instance, an A3 Project Manager may want to verify the isolation of his TVD with respect to other TVDs sharing the same resources.

9.4.2 Requirements

The use cases are depicted in Figure 9.20. The *Reasoner* is a component that will be included in the Management Component, while the *Enforcer* is a component that shall extend `libvirt` inside the Computing Nodes.

This subsystem also requires to discover the low-level configuration of the cloud infrastructure, i.e. of all Cloud Nodes and all the VMs running on Computing Nodes. For this functionality we rely on another subsystem, namely the Automated Validation described in Section 9.5.4.

9.4.2.1 Selected Use Cases

USE CASE UNIQUE ID	**/UC 470/ (Discovery)**
DESCRIPTION	The low-level configuration of the Cloud Nodes and the running VMs is fecthed from the Management Component. This use cases is detailed in /UC 520/, in Section 9.5.4.

USE CASE UNIQUE ID	**/UC 480/ (Input model)**
DESCRIPTION	The Management Component provides the system model as input to the reasoner.
ACTORS	Management Component.
PRECONDITIONS	Management Component performed a discovery, cf. /UC 470/.
POSTCONDITIONS	The reasoner updates its internal model.
NORMAL FLOW	1. The Management Component fetches the configuration/state from a data collector. 2. The Management Component feeds the reasoner with the configuration.

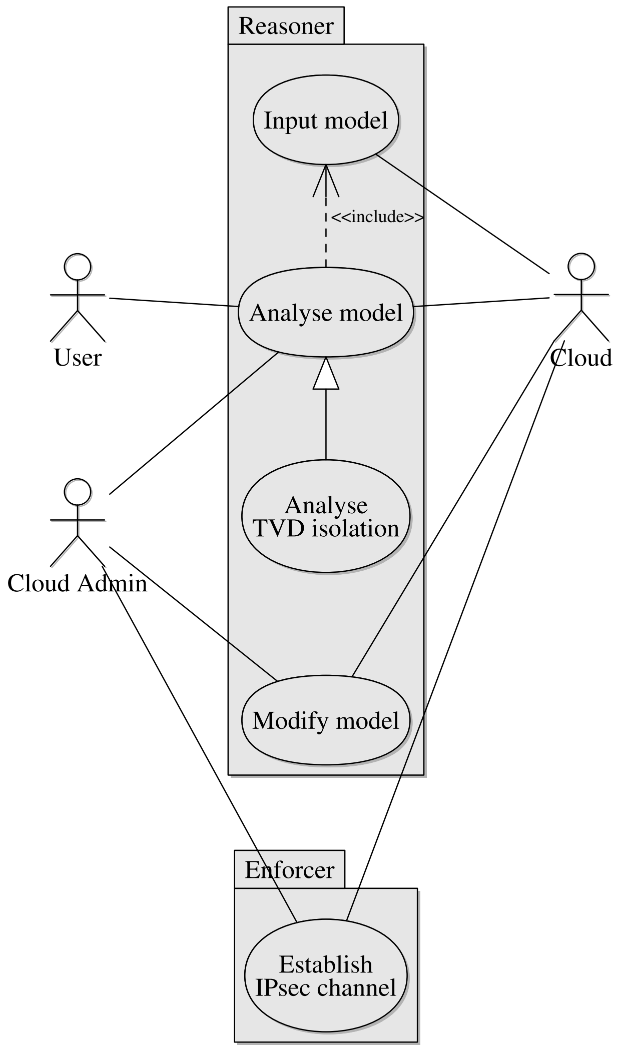


Figure 9.20: Use case diagram for the ontology-based reasoner (and enforcer) to check TVD isolation.

USE CASE UNIQUE ID	**/UC 490/ (Analyze model)**
DESCRIPTION	The reasoner performs an analysis of the model.
ACTORS	Management Component or Cloud Admin or User.
PRECONDITIONS	User is authorized to analyze the requested TVD (i.e. User owns that TVD).
POSTCONDITIONS	None.
NORMAL FLOW (TVD ISOLATION)	1. The Management Component requests the TVD isolation analysis for a specific TVD. 2. The reasoner performs the analysis to check TVD isolation. 3. Optionally, the result of this analysis can be used by the Management Component (e.g., by the Scheduler) to make decisions under some circumstances (e.g., creation of a new VM instance in a TVD).
ALTERNATIVE FLOW (TVD ISOLATION/CLOUD ADMIN)	1. The Cloud Admin, via Management Console, may request the analysis of specific system configurations (e.g., to make simulations).
ALTERNATIVE FLOW (TVD ISOLATION/USER–OPTIONAL)	1. The User, via Management Console, can request to check the isolation of his TVD.

USE CASE UNIQUE ID	**/UC 500/ (Modify model)**
DESCRIPTION	The system model is modified.
ACTORS	Management Component or Cloud Admin.
PRECONDITIONS	None.
POSTCONDITIONS	The reasoner updates its internal model.
NORMAL FLOW (NEW VM IN A TVD)	1. In the model, a new VM instance is added to a TVD and deployed on a Computing Node (i.e. a physical host).
ALTERNATIVE FLOW (NEW IPSEC CHANNEL)	1. In the model, a new IPsec channel is established between two Computing Nodes, and the network of a TVD is “deployed” onto this channel.

USE CASE UNIQUE ID	**/UC 510/ (Establish IPsec Channels)**
DESCRIPTION	An IPsec channel is established between two Computing Nodes (i.e. physical nodes).
ACTORS	Management Component or Cloud Admin.
PRECONDITIONS	None.
POSTCONDITIONS	An IPsec channel is established between two Computing Nodes.
NORMAL FLOW	1. The actor requests the establishment of an IPsec channel between two hosts (i.e. the Computing Nodes) to the infrastructure management. 2. The infrastructure management connects to the two hosts and instructs `libvirt` to setup the IPsec channel. Note: `libvirt` will be modified to support IPsec channel between hosts. The behaviour of the infrastructure management will be better specified in a future report.

9.4.2.2 Demo Storyboard

The following story shows how the ontology-based reasoner (and enforcer) can be used to: (1) verify the correct deployment of a TVD; (2) assist the administrator/infrastructure to guarantee isolation while modifying the structure of the TVD (e.g. adding a new VM to the TVD).

1. VM$_{A1}$ and VM$_{A2}$ in TVD$_{Aqua}$ are running on the Computing Node X (i.e. the same physical host X).

2. The reasoner takes as input a model of the system from the cloud infrastructure (cf. /UC 480/).

3. The reasoner is run in order to verify that the TVD$_{Aqua}$ is properly isolated (cf. /UC 490/).

We now begin a simulated phase:

4. User wants to add a new VM$_{A3}$ to the TVD$_{Aqua}$. We assume that VM$_{A3}$ will run on the Computing Node Y, distinct from X.

5. We (possibly manually) feed the reasoner with this new information (cf. /UC 500/, normal flow), run the reasoning (cf. /UC 490/) and discover that the channel between X and Y must be secured, and that now it is not secure.

6. We (possibly manually) add an IPsec channel between X and Y in the model (cf. /UC 500/, alternative flow), re-run the reasoning (cf. /UC 490/) and check that in this solution the TVD is now isolated.

Finally, we go back to the real world:

7. Cloud Admin issues a command to `libvirt` at X and Y to setup an IPsec channel between the Computing Nodes (cf. /UC 510/).

8. User adds a new VM$_{A3}$ to the TVD$_{Aqua}$ on Y.

9. We go back to steps 2-3 and verify on the real world that the TVD is properly isolated.

Note that step 7 is optional, in the sense that it could be automatically performed by the Management Component during the creation of VM$_{A3}$ (step 8).

9.4.3 Architecture

9.4.3.1 High-level Design

The high level architecture is depicted in Figure 9.21.

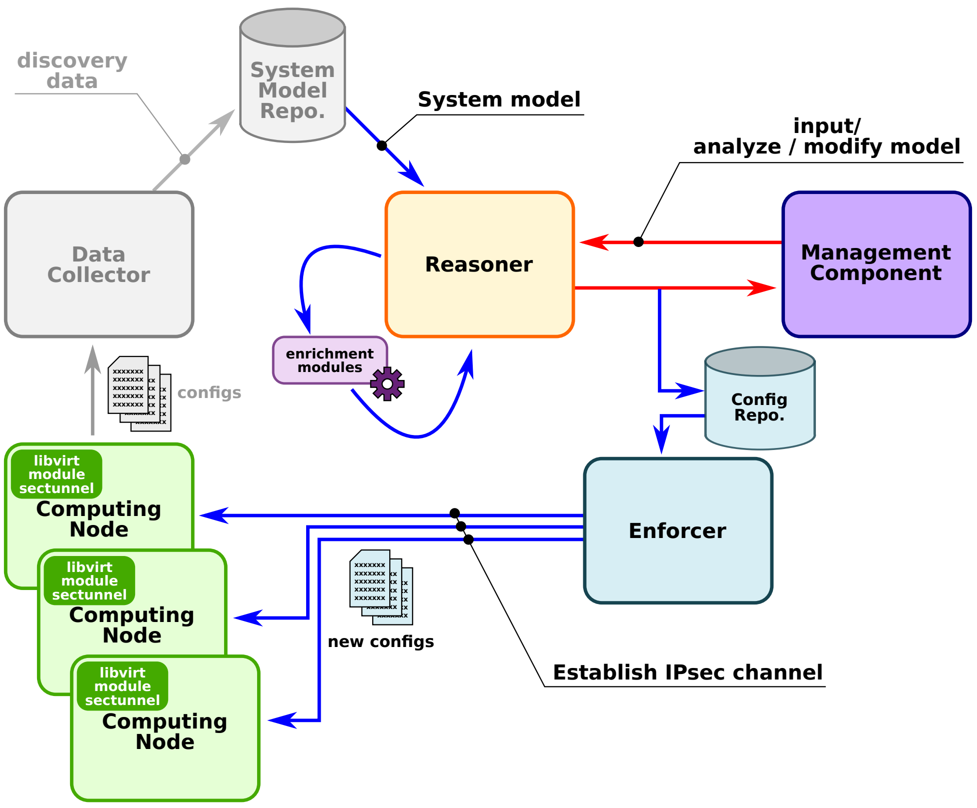


Figure 9.21: High-level architecture of the Ontology-based Reasoner.

9.4.3.1.1 Data Collector. It collects data from the infrastructure and makes it available for the components/subsystems that need it. This component is developed within the Automatic Audit subsystem, cf. Section 9.5.4. The “interface” between the Data Collector and the Reasoner is represented by a common format for the infrastructure, virtual, service and security models, that we collectively refer to as *System Model*.

9.4.3.1.2 Reasoner. This is the core of the Ontology-based Reasoner and allows automatic analysis on the system model, based on ontology reasoning. For details on the underlying ontology we refer to D2.3.1, Chapter 7.

The Reasoner functionalities include:

- *Input model*: to read the system model and map and import it in the internal ontology (cf. Section 9.4.3.2.1).
- *Analyze model*: to run an analysis on the current ontology (cf. Section 9.4.3.2.2).
- *Modify model*: for the Cloud Admin to manually alter the loaded ontology in order to perform other analysis.

Given a system model describing a set of TVDs and the underlying physical infrastructure, we support two specific analysis:

- *Verify TVD isolation*: outputs a list of possible network flows between TVDs.
- *Find Secure Tunnels*: outputs a list of required secure tunnels to avoid network flows between TVDs.

Sometimes the input model is not sufficiently detailed for the current analysis, thus it is necessary to add information (that can be either automatically computed, retrieved from the Management Component or requested to the Cloud Admin). For this reason, the Reasoner supports the addition of modules called *Enrichment Modules*.

Whenever an analysis outputs new configurations for the cloud infrastructure, such configurations are stored in a central *Configuration Repository*, that can be accessed by other components, e.g. the Enforcer.

The Reasoner does not directly interact with the Data Collector or the Enforcer, whilst the "interface" among these components is represented by a shared common format for the system model and the configuration repository. This design choice has been made to simplify future development and extensions.

9.4.3.1.3 Enforcer. It is a component able to reconfigure Cloud Nodes, in particular to let them establish new IPsec channels. The Enforcer reads configurations from the *Configuration Repository* and reconfigures Cloud Nodes via `libvirt`.

9.4.3.1.4 `libvirt` Module for Secure Tunnels. This is a module that extends `libvirt` to support the establishment of secure tunnels via IPsec.

9.4.3.2 Sequence Diagrams

The following sequence diagrams implement two use cases defined in Section 9.4.2 (specifically the *Input model* and the *Analyze model* use cases). The functions used in the sequence diagrams comes from the API discussed in Section 9.4.4.

9.4.3.2.1 Input model (Figure 9.22). The Management Component interacts with the Data Collector to collect the system model. When the data is available, the Management Component notifies this to the Reasoner.

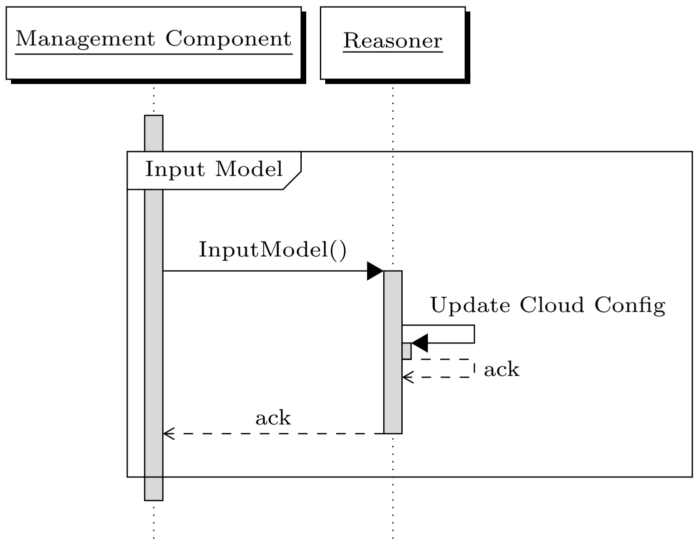


Figure 9.22: Sequence diagram for Ontology-based reasoner 'Input Model' use case.

9.4.3.2.2 Analyze Model. In the normal flow of the use case (Figure 9.23), the Management Component requests the Reasoner to check the cloud configuration or a specific property. An example can be TVD isolation: in this case the functions *RequestCheckConfig()* and *CheckConfig()* will accept the TVD identifier as parameter. These actions can be triggered within the Management Component, e.g. when a new VM instance must be started. The sequence of actions defined for the normal flow are a subset of actions for the alternative flows of this use case.

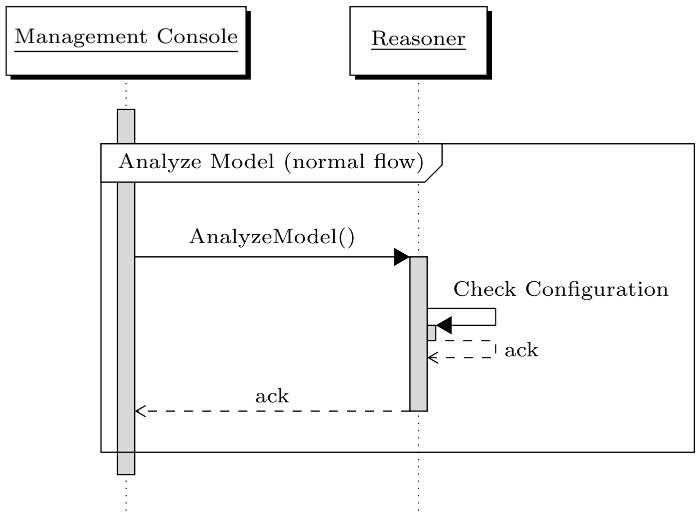


Figure 9.23: Sequence diagram for Ontology-based reasoner 'Analyze Model (Normal Flow)' use case.

In one alternative flow (Figure 9.24), the Cloud Admin interacts with Management Component to request the the checking of the cloud configuration or of a specific property.

In the other alternative flow (Figure 9.25), the Project Manager interacts with Management Component to request the checking of the isolation of its TVD.

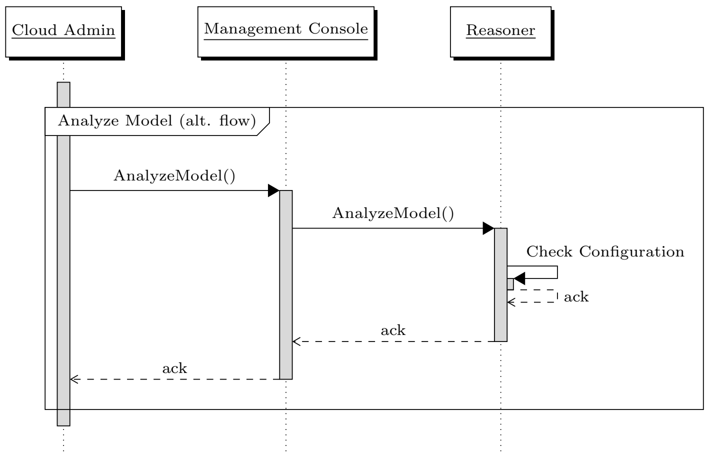


Figure 9.24: Sequence diagram for Ontology-based reasoner 'Analyze Model (Alternative Flow with Cloud Admin)' use case.

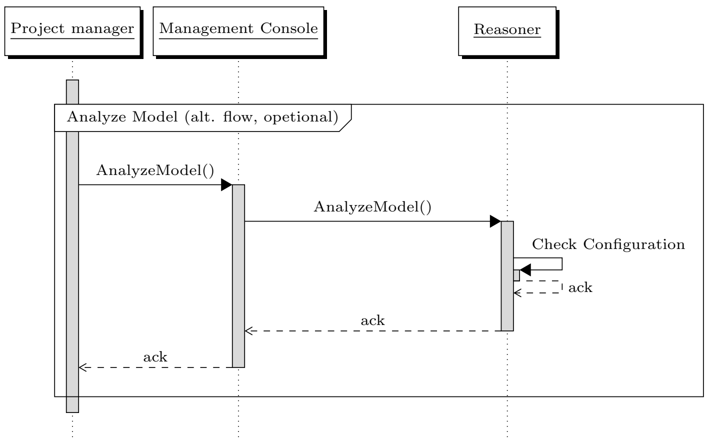


Figure 9.25: Sequence diagram for Ontology-based reasoner 'Analyze Model (Alternative Flow with Project Manager, optional)' use case.

9.4.4 API

9.4.4.1 Reasoner

We present a preliminary API for the Reasoner that exposes its main functionality. The API is preliminary in the sense that we are investigating the possibility to implement notifications among our components and the Management Component following the event/listener paradigm.

InputModel()

```
void ⇐ InputModel (void)
```

Description.
The `InputModel` function is called by the Management Component when the Reasoner should update the internal ontology according to the system model.

AnalyzeModel()

```
result ⇐ AnalyzeModel (query)
```

Description.
The `AnalyzeModel` function is called by the Management Component to run an analysis. The `query` parameter defines the specific analysis and in our proof of concept implementation it shall be chosen among a set of prefixed queries. The `result` depends on the actual analysis and it is expected to be in the final format required by the caller (specific modules may take care of the translation to the final format). Finally, we note that this function may have side effects, for instance a new configuration may be written in the configuration repository.

`ModifyModel()`

```
void ⇐ ModifyModel (modelDiff)
```

Description.
The `ModifyModel` function is called by the Management Component (usually the Management Console after request from the Cloud Admin) to modify the internal ontology without reading it from the system model.

9.4.4.2 Secure Tunnelling in `libvirt`

We present a proposal to extend the `libvirt` XML format to support secure tunnelling via IPsec.

9.4.4.2.1 Tunneling Networks. To ensure the separation of the information flows which are present on the same connection infrastructure, L2-technologies like VLAN may be used. In a virtual environment, if domains are running on the same physical host or are running on L2-adjacent hosts, they can communicate at L2 level, hence an efficient separation (e.g. VLAN) may be ensured.

When domains are running on different hosts which are not L2-adjacent, an L2 tunnel over L3 network is necessary. Tunnel employment makes possible, thanks to the encapsulation, the propagation of Ethernet frames via routed network as if the bridges where L2-adjacent.

An example of XML definition file that describe a virtual network which use a tunnel is showed below (Listing 9.4.1). The `<tunnel>` elements has been added to the `<network>` part of the standard `libvirt` configuration. This element specifis that a tunnel have to be used for communication with domains which are not directly connected through a bridge. More tunnels are possible, once for each physical host to be reached.

```
<network>
    <name>TVD_Aqua</name>
    <uuid>3e3fce45...</uuid>
    <bridge name="virbr0" />
    <forward mode="nat" dev="eth0"/>
    ...
    <tunnel name="sectun0" />
</network>
```

Listing 9.4.1: Virtual Network XML definition

9.4.4.2.2 Secure Tunnels. The `<tunnel>` element defines a new driver called *sectunnel* which describes a generic secure tunnel that will be established using several technologies.

Note that when the two domains connected are executed on different physical nodes, the usage of secure connections is necessary to integrity and the confidentiality of the data transferred. In the other case, the confidentiality is provided by the isolation guaranteed by the hypervisor.

In the following example, a tunnel protected with IPsec is presented. The configuration file defines a `name` for the tunnel, a `uuid` and then the tunnel `type` together with all the additional information necessary to create it (which may depend on the type of the tunnel).

```
<tunnel>
    <name>sectun0</name>
    <uuid>8b7fd1b0-4463-43b7-8b6e-8006344aeb66</uuid>
    <type>ipsec</type>
    <ipsec>
        <sa>
            <!-- here the Security Association definition -->
        </sa>
        <sp>
            <!-- here the Security Policy definition -->
        </sp>
    </ipsec>
</tunnel>
```

Listing 9.4.2: Secure Tunnel XML definition

9.4.4.2.3 Hooks Definition. The security of the TVD framework may be enhanced by applying SELinux policies to the virtual domains (cf. sVirt). Such a procedure may be executed automatically by defining a proper hook.

9.5 Automated Validation of Isolation of Cloud Users

Authors:
Sören Bleikertz, Christian Cachin, Thomas Groß, Michael Osborne (IBM)

9.5.1 Overview

9.5.1.1 Description

SAVE (Security Assurance for Virtual Environment) is a tool developed at IBM research for extracting configuration data from multiple virtualization environments, transforming the data into a normalized graph representation, and subsequent analysis of its security properties. IBM will integrate and adapt this technology for the demonstrator based on OpenStack, in order to validate isolation of cloud users.

9.5.1.2 Goals (security/privacy/resilience)

- Isolation of Tenants

  Description: The automated audit mechanism will validate that isolation of different tenants in the cloud infrastructure is given by analyzing the current configuration.

  Techniques/research problems: An information flow analysis on the virtualized infrastructure topology will be employed that forms the basis for a isolation breach diagnosis.

  Assumptions: The automated validation requires access to the configuration information on the physical nodes or through a central management interface. Perhaps, the OpenStack infrastructure itself can be extended and used to extract these information.

9.5.1.3 Required external components (relationship with Activity2)

None

9.5.1.4 Relationship with Activity3

None

9.5.2 Requirements

The automated audit can validate an isolation security goal against the current cloud infrastructure configuration. We share the use cases: /UC 480/, /UC 490/, and /UC 500/ with the ontology-based reasoner.

9.5.2.1 Selected Use Cases

USE CASE UNIQUE ID	**/UC 520/ (Discovery)**
DESCRIPTION	The audit tool will obtain the low-level configuration of the Cloud Nodes and their VMs from the Management Component.
ACTORS	Management Component and Cloud Nodes.
PRECONDITIONS	
POSTCONDITIONS	The complete low-level configuration of the Cloud Infrastructure was obtained and provided to the audit tool.
NORMAL FLOW	1. Audit tool will query the Cloud Infrastructure for the low-level configuration. 2. Configuration is provided to the audit tool.

USE CASE UNIQUE ID	**/UC 530/ (Isolation Analysis)**
DESCRIPTION	Based on a security policy given by the Cloud Admin, the isolation of a User on the Cloud is analyzed and validated with regard to the policy. Isolation can be described for storage, network, and compute resources.
ACTORS	Cloud Admin
PRECONDITIONS	A security policy describing the isolation goals is given. The Discovery was completed successfully.
POSTCONDITIONS	The isolation described in the policy is either validated or a violation alert will be given.
NORMAL FLOW	1. Security policy is read by the audit tool. 2. The policy is validated against the configuration data obtained in the Discovery. 3. Security policy is either satisfied or a violation will be shown.

9.5.2.2 Demo Storyboard

With use case /UC 520/, the auditor can obtain the current configuration of the cloud infrastructure. The audit tool will read the discovery data and update its internal model according to /UC 480/. A high-level security goal, such as zone isolation, will be validated against the model by the audit tool (cf. /UC 530/ and /UC 490/). In case a violation of the security goal is found, the audit tool will provide an example for such a security breach. Changes to the configuration will be reflected in model according to /UC 500/. Furthermore, the audit tool can simulate dynamic aspects of the infrastructure, such as virtual machine migrations, which will affect the internal model.

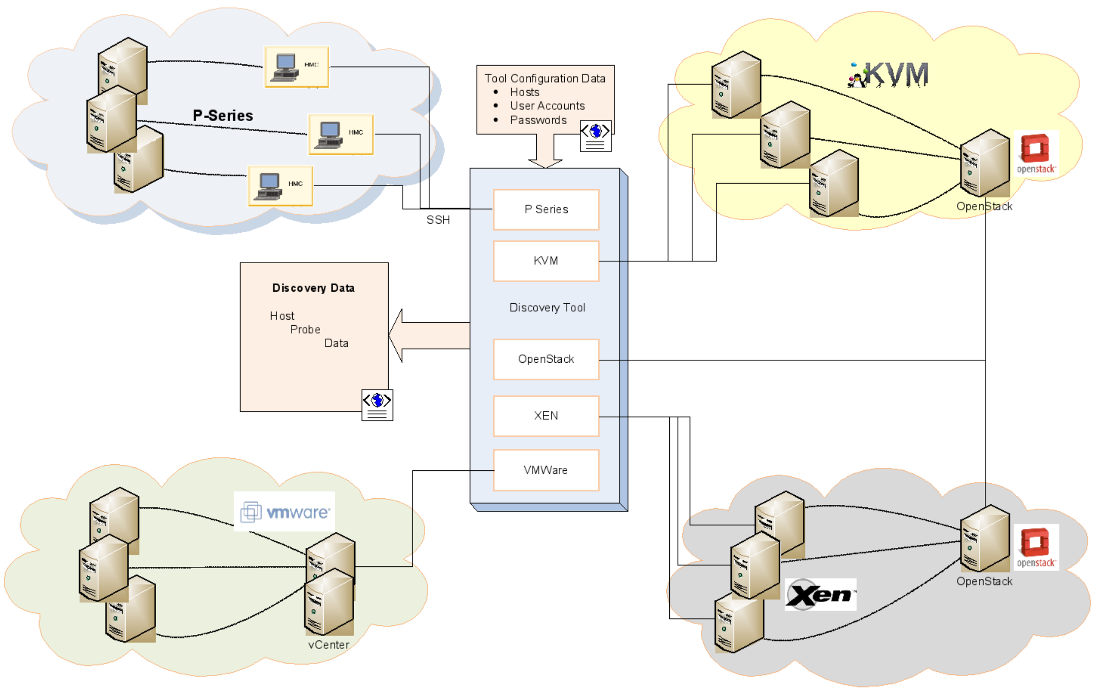


Figure 9.26: Architecture Overview of Discovery Component

9.5.3 Architecture

9.5.3.1 High-level Design

SAVE is structured into two components: *Discovery* and *Analysis*.

9.5.3.1.1 Discovery The data discovery phase is used to collect virtualization data from a number of heterogeneous environments (cf. Figure 9.26). It is configured with the set of hosts to query and basic authentication information required to access the data. The tool uses simple heuristics to identify the environment in which each host is situated. Based on the environment (HMC, VMware, XEN, libvirt) the appropriate probe is selected. The tool outputs a single XML file containing all of the virtualization information that was discovered. This XML file is used as input into the data analysis components.

9.5.3.1.2 Analysis The analysis component takes discovery XML format as input, in addition to a specification of traversal rules and a security policy (cf. Figure 9.27). The traversal rules are formulated in XML and specify the information flow and trust assumptions about elements of the virtualized infrastructure in general. The security policy is specified in a logical term language called VALID, which expresses attack states that violate the high-level security goals, in a nutshell. VALID is language developed with a formal methods background and based on the AVISPA Intermediate Format (IF) and ASlan, two languages that widely used in model checking.

For the validation of the discovered infrastructure against the security policy SAVE will compile problem statements for model checkers in their respective language, such as IF, ASlan or First-Order Logic (FOL). It also takes proprietary output format of the model checkers as feedback and evaluates their output with respect to the realization model to find alarm states.

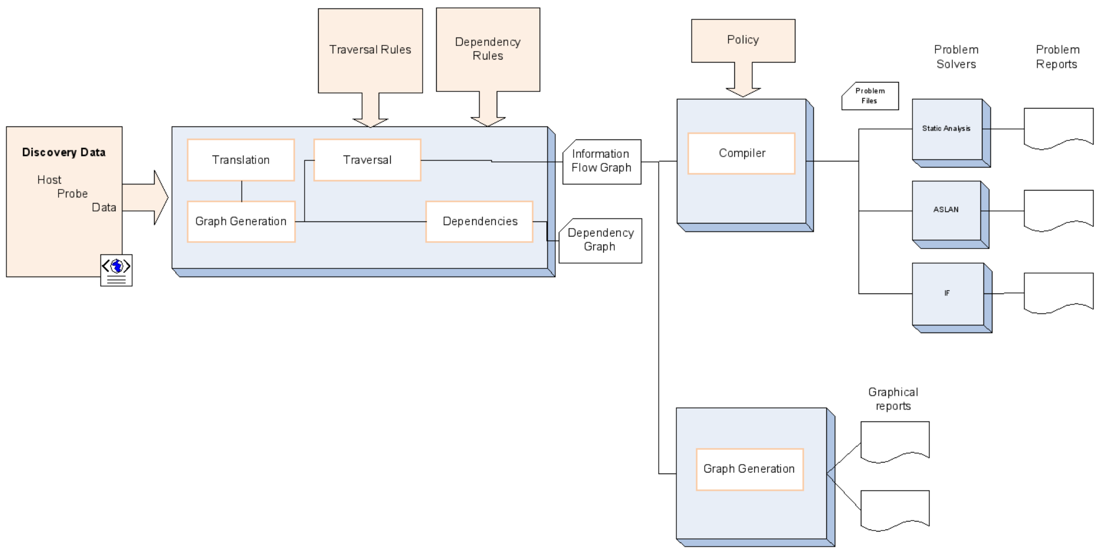


Figure 9.27: Architecture Overview of Analysis Component

The general persistent output of the SAVE analysis may be textual, as fault logs, or a standard graph format (GEXF), in order to render big-picture views on the topology and fine-grained views on problem areas for diagnosis.

9.5.3.2 Sequence Diagrams

9.5.3.2.1 Discovery The sequence diagram for the Discovery component is illustrated in Figure 9.28. We distinguish between three sub-phases: Setup, Discovery, and Output. In *Setup*, the cloud administrators configures the validation program SAVE, i.e., he provides the hosts (individual compute nodes or the management host) that should be discovered and corresponding access credentials. *Discovery* concerns the use case /UC 520/ and performs an iterative discovery over the hosts while trying different discovery probes. The gathered data from the probes are stored in an XML file and returned to the administrator in sub-phase *Output*.

9.5.3.2.2 Analysis Figure 9.29 illustrates the sequence diagram for the Analysis component. It is also structured into three sub-phases: Setup, Analysis, and Output. In *Setup* the admin provides the previously obtained discovery XML file as well as a security policy that will be used for the analysis. In the *Analysis*, the discovery XML file is transformed into an internal model (cf. /UC 480/), an information flow graph is derived, and the validation of the policy is performed (cf. /UC 490/ and /UC 530/). A report of the analysis is generated and returned to the administrator in *Output*.

9.5.3.3 Low-level Design

Language: Audit technology is written in Java and can be packaged as a jar file. A simple wrapper for providing an API can be written in a language such as Python. The policy is specified in a language designed by IBM. **Existing SW:** Our prototype is mainly used for VMware based virtual environments, but other virtualization technologies such as Xen and LibVirt (KVM) are supported as well.

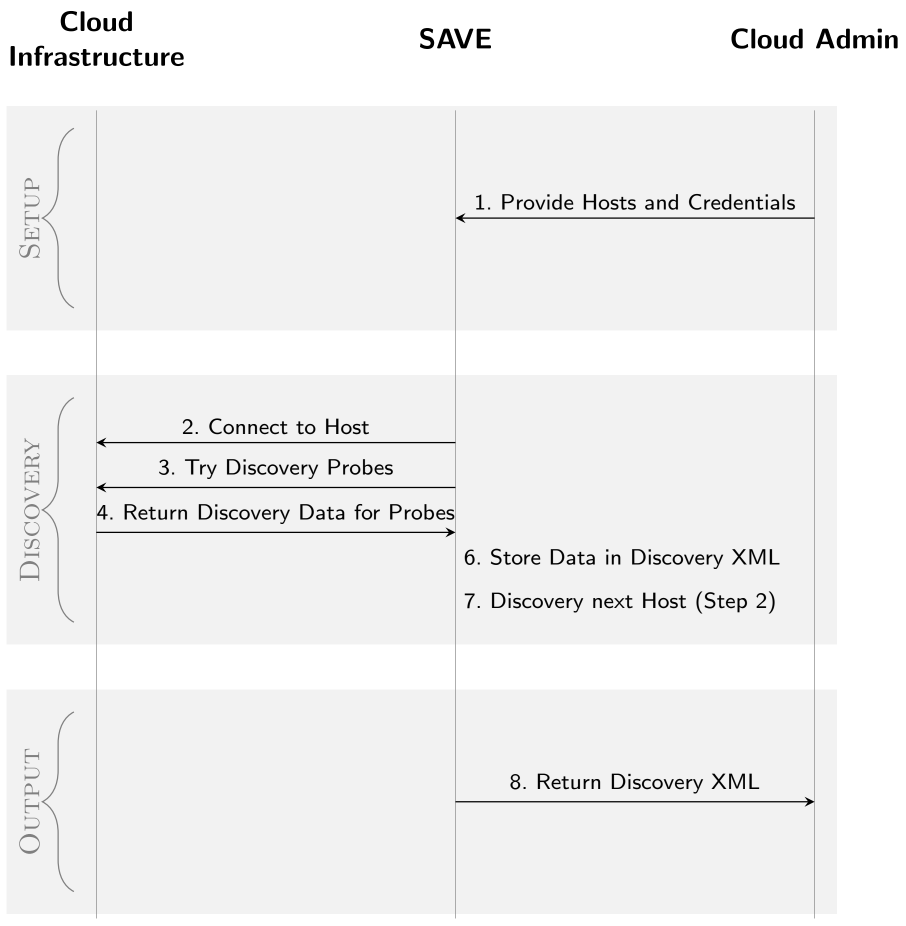

Cloud Infrastructure
SAVE
Cloud Admin
SETUP
1. Provide Hosts and Credentials
DISCOVERY
2. Connect to Host
3. Try Discovery Probes
4. Return Discovery Data for Probes
6. Store Data in Discovery XML
7. Discovery next Host (Step 2)
OUTPUT
8. Return Discovery XML

Figure 9.28: Sequence Diagram for Discovery

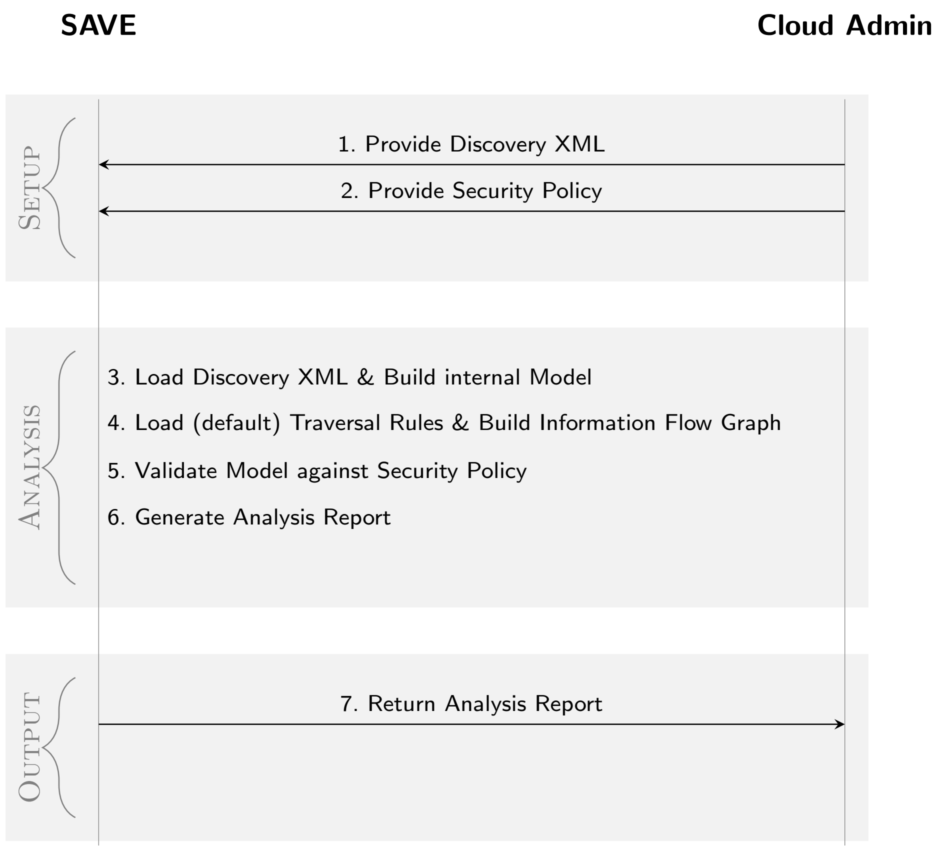

SAVE
Cloud Admin
SETUP
1. Provide Discovery XML
2. Provide Security Policy
ANALYSIS
3. Load Discovery XML & Build internal Model
4. Load (default) Traversal Rules & Build Information Flow Graph
5. Validate Model against Security Policy
6. Generate Analysis Report
OUTPUT
7. Return Analysis Report

Figure 9.29: Sequence Diagram for Analysis

9.5.4 API

We following API is considered public, i.e., accessible not only by internal components, but restricted. Typically a cloud administrator should be able to analyze the complete infrastructure, while cloud consumers might only be able to analyze a subset.

`Analyze()`

```
Analysis ID ⇐ Analyze (Security policy written in VALID)
```

Description.
Analyze infrastructure with given policy (i.e. isolation of tenants policy)

`Query()`

```
Status ⇐ Query (Analysis ID)
```

Description.
Query analysis status

`Results()`

```
URI for report ⇐ Results (Analysis ID)
```

Description.
Obtain diagnosis and report about possible isolation breaches

Part III

Appendix

Appendix A

First-round Analysis of Cloud Frameworks

A.1 Template for the Analysis of Cloud Frameworks

Summary
A brief summary of the framework

Core use of the framework
Essentially, why the framework has been designed?
Who is the main target?

Support and Community
Is the framework commercially used (in a large scale)?
Is there an industry player backing it?
How alive is the open source community?

Cloud model
Is it a IaaS? Or What?
Does it have specific components for, e.g., storage?

Installation How easy is (did you find) it to install?
On which platform have you installed it?
Which other platform are supposed to work?

Running applications
How easy is to run applications/create VMs instances?
Is there any GUI/web interface?

Virtualization environments and system configurations
Which VMM are supported?
Which storage system is used?
What about networking?

Programming languages
How is the framework written?

Clouds of clouds
Does it support hybrid clouds?
Which public clouds are actually supported?

Standard interfaces
Which API does it provides? Is it standard?
Which standard APIs are also supported/used?

Distribution license
Under which licence is it distributed?

Other information
Any other interesting point?
What TClouds could add to this project?

A.2 Eucalyptus

Summary
Eucalyptus' primary purpose is to build and manage private clouds within single organizations. It provides Infrastructure as a Service (IaaS) for VM-based computing ("Eucalyptus") and storage ("Walrus"). The external APIs for users of the infrastructure are mostly compatible with the services offered by Amazon. Eucalyptus doesn't provide any support for managing virtual machine images running on other providers (Cloud-of-Cloud infrastructure).

A typical Eucalyptus installation consists of several services working together. The "Node Controller" is responsible for managing the virtual macines on a virtual node. The "Cluster Controller" is responsible for scheduling and monitoring the execution of virtual machines on a set of cluster nodes. The "Walrus" service provides a storage facility which is compatible with Amazon S3 and hosts the virtual machine images for distribution to the individual nodes. Finally the "Cloud Controller" coordinates all the previously listed services and provides the interfaces for the end-user.

Core use of the framework

- *Essentially, why the framework has been designed?*
  The framework was designed to provide an alternative to the Cloud services offered by Amazon's EC2 and S3.
- *Who is the main target?*
  The main target are small/medium companies that want to build their own cloud infrastructure for their private use.

Support and Community

- *Is the framework commercially used (in a large scale)?*
  There are some well known organizations that seem to use it, e.g. NASA, Unisys, Trend Micro. We didn't find any numbers on how many nodes there are currently running, except for the public accessible Eucalyptus Community Cloud, which runs on 60 nodes.
- *Is there an industry player backing it?*
  Eucalyptus Inc., which sells the commercial editions of the cloud platform and has some cooperation with smaller companies that provide additional services like training and consulting.

- *How alive is the open source community?*
  Development happens mostly on launchpad[1]. The launchpad statistics say, that there were 99 commits to the source code repository by 10 different people in the month of evaluation. So it seems to be only a small community which is working actively on the project.

Cloud model

- *Is it a IaaS? Or What?*
  It provides the infrastructure for setting up virtual machines running Linux in different configurations. No applications specific services (except the S3 compatible storage called "Walrus") are offered, so it can be characterized mostly as Infrastructure as a Service (IaaS).
- *Does it have specific components for, e.g., storage?*
  Yes, it offers a Storage service "Walrus" that is mostly compatible to Amazon's S3. Unfortunately, there is no built-in support for redundancy or distributed storage. The Walrus service is used as a the central repository and storage for virtual machine images.

Installation

- *How easy is (did you find) it to install?*
  The installation and initial configuration works quite easy. Major issues arised because the network environment for testing wasn't entirely under control of the Eucalyptus infrastructure (e.g. external DHCP servers, no VLANs) and required customized firewall rules.
- *On which platform have you installed it?*
  Ubuntu LTS 10.04
- *Which other platform are supposed to work?*
  Pre-built packages are available for every major Linux distribution, sometimes with extra features for integration with additional services offered by the distributor (e.g. Ubuntu Enterprise Cloud). Eucalyptus Systems Inc. also offers to test drive the software from a user's perspective on a small cluster available to the community.

Running applications

- *How easy is to run applications/create VMs instances?*
  Quite easy. Disk Images and kernels can be packaged, uploaded and started on the command line using the euca2ools, which is available for every major Linux distribution and just a set of Python scripts.
- *Is there any GUI/web interface?*
  Yes, there is a Firefox Plugin "Hybridfox", which shows an quick overview of running virtual machine instances, what disk/kernel images are available and allows starting/stopping instances in various configurations, as well as SSH integration to connect to running instances. Eucalyptus features a Web-Interface for many daily administrative tasks, like adding user accounts and managing resources limits.

[1]https://launchpad.net/eucalyptus

Virtualization environments and system configurations

- *Which VMM are supported?*
  Virtual machines are handled by "libvirt". Xen and Linux KVM are actively supported, with KVM being the recommended choice. The free open-source edition only supports Linux as guest operating system. Commercial version can also run Windows Servers and use VMware on the virtualization layer.

- *Which storage system is used?*
  Disk images and other files for the infrastructure are managed on the node running the Walrus service. They are stored in the Linux filesystem and made available via an interface that is mostly compatible with Amazon's S3.

- *What about networking?*
  Eucalyptus wants most of the networking under it's own control. There are different networking models available: SYSTEM, STATIC and MANAGED. Our tests with Eucalyptus 1.6.2 from Ubuntu Lucid showed that the simpler SYSTEM and STATIC modes don't seem to work properly in some cases and are hard to manage. The MANAGED mode where Eucalyptus does most of the networking configuration (DHCP, VLANS, iptables...) should be used if possible.

Programming languages

- *How is the framework written?*
  The services are mostly written in Java and Groovy. Some system-level components were implemented in C and exposed as web services using the Apache Axis2C server.

Clouds of clouds

- *Does it support hybrid clouds?*
  Only as a client. There is no support for moving virtual machines to external cloud service providers or managing machines running on another cloud platform.

- *Which public clouds are actually supported?*
  N/A

Standard interfaces

- *Which API does it provides? Is it standard, which standard APIs are also supported/used?*
  From a user's perspective, Eucalyptus works the same way as Amazon's Web Services for all basic features. Unfortunately there are some minor API differences, like for example different escaping of file paths in Walrus compared to Amazon S3. So in practice some tools need minor patching to work reliable with Eucalyptus instead of Amazon's implementation of the services.

  Other features, like the user management, monitoring and administration work entirely different compared to Amazon's services.

Distribution license

- *Under which licence is it distributed?*
  The open source edition is distributed under the terms of GPLv3 (no earlier version). Some minor parts have a simplfied BSD-License. A commercial cloused-source, version with more features is also available.

Other information

- *Any other interesting point? What TClouds could add to this project?*
  The cloud-of-clouds components and distribution needs to be added, e.g. by combining with another solution like OpenNebula.

A.3 OpenNebula

Summary [2]
OpenNebula is an open-source project that intends to build a comprehensive, scalable and adaptable tool to manage distributed infrastructures as found in cloud computing. It has been developed to address the requirements of multiple business use cases in the context of some research projects in cloud computing. It is being used as an open platform for innovation in several research projects and also as an industrial production-ready tool.

To address the requirements of business use cases (Hosting, Telecom, eGovernment, Utility Computing, etc.), OpenNebula's design principles are:

- Openness of the architecture, interfaces, and code
- Adaptability to manage any hardware and software combination, and to integrate with any product and service in the cloud and virtualization ecosystem
- Interoperability and portability to prevent vendor lock-in
- Stability for use in production enterprise-class environments
- Scalability for large scale infrastructures
- Standardization by leveraging and implementing standards

OpenNebula is open and flexible, and fits into the data center environment to build any type of IaaS cloud. It manages storage, network, virtualization, monitoring, and security technologies to enable the dynamic placement of multi-tier services (groups of interconnected virtual machines) on distributed infrastructures, combining both data center resources and remote cloud resources, according to allocation policies. OpenNebula can manage a virtual infrastructure in a cluster (private cloud), provides interfaces to expose its functionality for virtual machine, storage and network management (public clouds) and can combine local infrastructures with public cloud-based infrastructures (hybrid clouds). Its features include:

- Cloud management: It contains features for cloud management of users, images, services, infrastructures, storage, virtual machines and networks, and a policy-oriented matchmaker and workload allocator (scheduler). It is capable of combining private and public clouds and federating different clouds to build a hierarchy (scalability).

[2]Compiled from the project website: `http://www.opennebula.org`

- Cloud integration: Open, flexible and extensible architecture, interfaces and components. It offer also an infrastructure abstraction layer independent of its underlying services for virtualization, networking, security and storage.
- Production environment: Scalability and performance tested on large infrastructures. It uses an authentication framework based on passwords, ssh/RSA keypairs or LDAP. It performs external and internal communications through SSL, secure multi-tenancy, or isolated networks.

OpenNebula was established as a research project in 2005 by Ignacio M. Llorente and Rubn S. Montero in the Complutense University of Madrid. It had its public release in 2008, and now operates as an open source project managed by C12G, a privately-held, self-funded, spin-off research lab and technology provider.

Core use of the framework

- *Essentially, why the framework has been designed / Who is the main target?*
  OpenNebula aims to lead innovation in enterprise-class cloud computing management. It is intended to be the most-advanced, highly-scalable and adaptable software toolkit for cloud computing management.

  The project owners intend to assure the stability and quality of their software and to collaborate with demanding users of cloud management tools. They want to support an ecosystem of open-source components around the project as well as user and developer communities.

Support and Community

- *Is the framework commercially used (in a large scale)?*
  N/A
- *Is there an industry player backing it?*
  N/A
- *How alive is the open source community?*
  N/A

Cloud model

- *Is it a IaaS? Or What?*
  OpenNebula is built upon a hybrid, compositional, cloud model. It offers tools for managing private clouds, to offer public clouds from the managed ones, and to integrate public clouds with private clouds in a hierarchy of clouds.
- *Does it have specific components for, e.g., storage?*
  No.

Installation

- *How easy is (did you find) it to install?*
  OpenNebula is easy to install. It is done in two steps: front-end and node installation. The front-end is easily installed from the source using Scons (software construction tool) and has well documented special requirements. There is an Ubuntu distribution that contains all packages for this installation. The nodes require a virtual machine monitor (as Xen or KVM), Ruby, SSH, and a user to be used by the front-end scripts. In the nodes, installing and configuring the virtual machine monitor is usually more complicated than the OpenNebula related requirements.
- *On which platform have you installed it?*
  c.f. 4.1.1
- *Which other platform are supposed to work?*
  c.f. 4.1.1

Running applications

- *How easy is to run applications/create VMs instances? Is there any GUI/web interface?*
  It's easy also to run and use OpenNebula, mainly using the command line interface provided (script named `oneXXX`). Some examples:
  - Initialise OpenNebula (`$ one start`)
  - Create a virtual network, based on a template (`$ onevnet create`)
  - Add hosts to OpenNebula control (`$ onehost create`)
  - Start virtual machines (`$ onevm create`)

Virtualization environments and system configurations

- *Which VMM are supported?*
  OpenNebula can work with the following virtual machine monitors:
  - Xen
  - KVM
  - VMWare
- *Which storage system is used?*
  N/A
- *What about networking?*
  N/A

Programming languages

- *How is the framework written?*
  OpenNebula's core was developed in C++ and many modules are made in Ruby. Remote interfaces are implemented using XML-RPC and it offers interfaces for Shell scripts, Ruby and Java.

Clouds of clouds

- *Does it support hybrid clouds? Which public clouds are actually supported?*
  OpenNebula offers the possibility to combine private cloud with resources from one or several public cloud providers. The remote provider can be a commercial cloud service, such as Amazon EC2 or ElasticHosts, or a partner's private infrastructure running a different OpenNebula instance. It is fully transparent to infrastructure users. Users continue using the same private and public cloud interfaces. It is the infrastructure administrator who takes decisions about the scale out of the infrastructure according to infrastructure or business policies.

  To use public clouds OpenNebula needs a number of cloud service adaptors configured. Service adaptors are like drivers, and there are different adaptors for Amazon EC2 and ElasticHosts, although the same interface can be used to produce new adaptors as needed.

Standard interfaces

- *Which API does it provides? Is it standard?*
  OpenNebula provides the OpenNebula Cloud API (OCA) and implemented over it some standard interfaces, namely:
  - OGF OCCI - Open Grid Forum Open Cloud Computing Interface
  - Amazon EC2 - Elastic Compute Cloud
  - vCloud Express API
  - XML-RPC
  - Libvirt XML
- *Which standard APIs are also supported/used?*
  Other interfaces are implemented in OpenNebula to integrate and create hybrid clouds, namely:
  - Amazon EC2
  - Elastic Hosts

Distribution license

- *Under which licence is it distributed?*
  The OpenNebula is licensed for use under the terms of the Apache License, Version 2.0, with Copyrights (2002-2010) to OpenNebula Project Leads.

  It is open-source code, not open-core, which development is restricted to major contributors and acceptance of patches for bugfixes, features and documentation, are made through virtualization and cloud ecosystems.

Other information

- *Any other interesting point? What TClouds could add to this project?*
  OpenNebula incorporates a pluggable scheduler module. Two implementations currently exist, one using Haizea (Univ. Chicago), to support advance reservation of resources and queuing best-effort requests, and another from the OpenNebula project. The OpenNebula Basic Scheduler has an easy-of-use and extensible scheduling policy based in host requirements and rankings. It means that always a VM is created, it's possible to define some mandatory requirements to filter hosts that are not desirable and rank the remaining ones based in other expressions or heuristics. TClouds can extend the policies to include trust requirements.

A.4 OpenStack

Summary
OpenStack consists of two components: an object storage system called 'swift', which is similar to Amazon S3, and a compute system called 'nova' (similar to Amazon EC2). Swift is actually used in its current form at RackSpace (the second largest/popular public IaaS provider), and Nova was developed at NASA and will be deployed at RackSpace in the future to replace their current system. NASA currently uses Eucalyptus in their IaaS project Nebula, but developed Nova due to scalability issues with Eucalyptus. Details are in Section 4.1.2.

Core use of the framework

- *Essentially, why the framework has been designed? Who is the main target?*
  Open source replacement of EC2 and S3 that you can use to set up your own cloud (on your own hardware). One strong focus seems to be scalability (via non blocking asynchronous processing) and modularity.

Support and Community

- *Is the framework commercially used (in a large scale)?*
  Unknown

- *Is there an industry player backing it?*
  Yes. Rackspace, NASA, Intel, etc.

- *How alive is the open source community?*
  Although OpenStack only had two official releases, the code base is derived from RackSpace's production systems and NASA's new IaaS project. The community is organized around LunchPad, a opensource project management system popularized by Ubuntu Linux, and gained support by major vendors and startups in the cloud infrastructure space, such as RackSpace, NASA, Intel, and Opscode (founded by creators of Amazon EC2).

Cloud model

- *Is it a IaaS? Or What?*

- *Does it have specific components for, e.g., storage?*

Installation cf. 4.1.2

Running applications cf. 4.1.2

Virtualization environments and system configurations

- *Which VMM are supported?*
  Xen, KVM, QEMU, User Mode Linux Support, Hyper-V
- *Which storage system is used?*
  A storage system is compatible to Amazon S3 and is also developed within OpenStack.
- *What about networking?*
  cf. 4.1.2

Programming languages

- *How is the framework written?*
  The main programming language is Python. For Swift there are bindings for PHP, Python, Java, Ruby and .NET (C#).

Clouds of clouds

- *Does it support hybrid clouds?*
  No.
- *Which public clouds are actually supported?*
  N/A

Standard interfaces

- *Which API does it provides? Is it standard?*
  Its own API, Swift and Nova.
- *Which standard APIs are also supported/used?*
  - Subset of EC2
  - Subset of Rackspace API
  - Subset S3 API

Distribution license

- *Under which licence is it distributed?*
  Apache 2.0 license.

Other information

- *Any other interesting point?*
  OpenStack has multiple security features: Nova supports Amazon EC2's firewall concept of Security Groups, which allows scalable firewalling for virtual machines. Furthermore, the networking in Nova can be based on VLANs for strong isolation. RBAC is supported for the Nova API and ACLs are provided for the Swift object storage system. For a production-grade system like OpenStack's Swift, it is essential to have strong security in order to provide customer isolation.

- *What TClouds could add to this project?*
  Resilience, Privacy, Cloud of Clouds.

A.5 Nimbus

Summary

A IaaS framework for science that has been spawned by the Globus toolkit community ('the grid toolkit'). Nimbus clouds are offered by five Universities in the USA. The toolkit follows a linux-like approach where many small tools are loosely coupled to provide the look-and-feed of a cloud.

The documentation is good and (while complex) the installation seems to be straightforward.

Core use of the framework

- *Essentially, why the framework has been designed? Who is the main target?*
  Nimbus is a set of open source tools that together provide an "Infrastructure-as-a-Service" (IaaS) cloud computing solution. Its mission is to evolve the infrastructure with emphasis on the needs of science, but many non-scientific use cases are supported as well.

Support and Community

- *Is the framework commercially used (in a large scale)?*
  N/A

- *Is there an industry player backing it?*
  N/A

- *How alive is the open source community?*
  N/A

Cloud model

- *Is it a IaaS? Or What?*
  IaaS for arbitrary clouds. It also claims to support federation of multiple clouds (e.g. a private cloud augmented with Amazon EC2 resources).

- *Does it have specific components for, e.g., storage?*
  N/A

Installation

- *How easy is (did you find) it to install?*
  Medium: Since it follows the linux paradigm of many collaborating tools, it is non-trivial to understand. However, the documentation seems good and since it's growing since 2008 it may have reached some maturity.

- *On which platform have you installed it?*
  N/A

- *Which other platform are supposed to work?*
  N/A

Running applications

- *How easy is to run applications/create VMs instances?*
  N/A

- *Is there any GUI/web interface?*
  N/A

Virtualization environments and system configurations

- *Which VMM are supported?*
  It supports the Xen and KVM hypervisors via the `libvirt` API.

- *Which storage system is used?*
  N/A

- *What about networking?*
  N/A

Programming languages

- *How is the framework written?*
  N/A

Clouds of clouds

- *Does it support hybrid clouds?*
  There is something like a 'context handler' that allows to establish security contexts between different clouds. However, I expect that this will be fairly simple.

- *Which public clouds are actually supported?*
  N/A

Standard interfaces

- *Which API does it provides? Is it standard?*
  The APIs seem to be REST. It allows controlling EC2 instances.

- *Which standard APIs are also supported/used?*
  N/A

Distribution license

- *Under which licence is it distributed?*
  Open Source.

Other information

- *Any other interesting point?*

- *What TClouds could add to this project?*

Appendix B

List of Tools and Projects Referred

B.1 Open Source Cloud Frameworks

OpenNebula [opea] OpenNebula.org is an open-source project aimed at building the industry standard open source cloud computing tool to manage the complexity and heterogeneity of distributed data center infrastructures. Fully open source (not open core), thoroughly tested, customizable, extensible and with unique features and excellent performance and scalability to manage hundreds of thousands of VMs.

OpenStack [opeb] OpenStack is a collection of open source technology products delivering a scalable, secure, standards-based cloud computing software solution. OpenStack is currently developing two interrelated technologies: OpenStack Compute and OpenStack Object Storage. OpenStack Compute is the internal fabric of the cloud creating and managing large groups of virtual private servers and OpenStack Object Storage is software for creating redundant, scalable object storage using clusters of commodity servers to store terabytes or even petabytes of data.

Nimbus [nim] Nimbus is an open source toolkit that allows you to turn your cluster into an Infrastructure-as-a-Service (IaaS) cloud.

Eucalyptus [euc] Eucalyptus Systems delivers private cloud software. This is infrastructure software that enables enterprises and government agencies to establish their own cloud computing environments. With Eucalyptus, customers make more efficient use of their computing capacity, thus increasing productivity and innovation, deploying new applications faster, and protecting sensitive data, while reducing capital expenditure.

B.2 Testing Tools and Frameworks

Hudson [hud] Hudson monitors executions of repeated jobs, such as building a software project or jobs run by cron. Hudson provides an easy-to-use continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. Also allows monitoring executions of externally-run jobs, such as cron jobs and procmail jobs, even those that are run on a remote machine.

Selenium [sel] Selenium has many projects that combine to make a versatile testing system, including the followings.

Selenium Core is the original Javascript-based testing system. It's now used primarily as a component of Selenium Remote Control, but it can also be used as a pure Javascript/HTML testing system.

Selenium IDE is a Firefox add-on that makes it easy to record and playback tests in Firefox 2+. You can even use it generate code to run the tests with Selenium Remote Control.

Selenium Remote Control is a client/server system that allows you to control web browsers locally or on other computers, using almost any programming language and testing framework.

SIKULI [sik] Sikuli is a visual technology to automate and test graphical user interfaces (GUI) using images (screenshots). Sikuli includes Sikuli Script, a visual scripting API for Jython, and Sikuli IDE, an integrated development environment for writing visual scripts with screenshots easily. Sikuli Script automates anything you see on the screen without internal API's support. You can programmatically control a web page, a desktop application running on Windows/Linux/Mac OS X, or even an iphone application running in an emulator.

B.3 Public Cloud Services

Amazon EC2 [amaa] Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.

Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios.

Amazon S3 [amab] Amazon S3 is storage for the Internet. It is designed to make web-scale computing easier for developers.

Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

Microsoft HealthVault [hea] HealthVault is a free online service that stores your health records in a central location, then lets you use the information with online health tools to manage health conditions, create fitness plans, prepare for doctor visits, and more.

Bibliography

[AGS$^+$08] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An efficient implementation of trusted channels based on openssl. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, STC '08, pages 41–50, New York, NY, USA, 2008. ACM.

[amaa] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2.

[amab] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3.

[amac] Amazon Simple Storage Service (Amazon S3) API Reference. http://docs.amazonwebservices.com/AmazonS3/latest/API/.

[CCS$^+$11] Giovanni Cabiddu, Emanuele Cesena, Roberto Sassu, Davide Vernizzi, Gianluca Ramunno, and Antonio Lioy. The trusted platform agent. *IEEE Software*, 28:35–41, 2011.

[ER03] Albert Endres and Dieter Rombach. *A Handbook of Software and System Engineering*. Addison Wesley, 2003.

[euc] Eucalyptus Open Source. http://open.eucalyptus.com.

[eve] Read-After-Write Consistency in Amazon S3. http://shlomoswidler.com/2009/12/read-after-write-consistency-in-amazon.html.

[GPS06] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the first ACM workshop on Scalable trusted computing*, STC '06, pages 21–24, New York, NY, USA, 2006. ACM.

[hea] Microsoft HealthVault. http://www.healthvault.com.

[hud] Hudson – continuous integration system. http://java.net/projects/hudson.

[jen] Jenkins – an extendable open source continuous integration server. http://jenkins-ci.org/.

[Lin08] Linbit. DRBD: What is DRBD, 2008. http://www.drbd.org.

[MT09] Di Ma and Gene Tsudik. A new approach to secure logging. *Trans. Storage*, 5:2:1–2:21, March 2009.

[nim] Nimbus. http://www.nimbusproject.org.

[nov] NOVA Microvisor. http://os.inf.tu-dresden.de/~us15/nova/.

[opea] OpenNebula. http://www.opennebula.org.

[opeb] OpenStack. http://www.openstack.org.

[Opec] OpenStack Administration Guide. http://docs.openstack.org/.

[Oped] OpenStack. Openstack architecture. http://nova.openstack.org/service.architecture.html.

[Ope10] OpenStack. Easyapi, 2010. http://wiki.openstack.org/EasyApi.

[Ope11] OpenStack. Openstack rest api, 2011. http://wiki.openstack.org/OpenStackRESTAPI.

[Red10] RedHat. LVM administrator guide - edition 1, 2010. http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Logical_Volume_Manager_Administration/.

[s3f] s3fs - FUSE-based file system backed by Amazon S3. https://code.google.com/p/s3fs/wiki/FuseOverAmazon.

[SCG+03] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171, New York, NY, USA, 2003. ACM.

[sel] SeleniumHQ. http://seleniumhq.org.

[sik] Sikuli. http://www.sikuli.org.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems*, 2:159–176, May 1999.

[smb] SMB: The Server Message Block Protocol. http://ubiqx.org/cifs/SMB.html.

[Smi11] Paolo Smiraglia. [one-users] EC2 API PROBABLY BUGS. http://www.mail-archive.com/users@lists.opennebula.org/msg01451.html, 2011.

[tcg] Trusted Computing Group. http://www.trustedcomputinggroup.com.

[Tim09] Falko Timme. Using iSCSI on Fedora 10 (initiator and target), 2009. http://www.howtoforge.com/using-iscsi-on-fedora-10-initiator-and-target.

[tpm] Trusted Platform Module (TPM) Main Specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[xen] Xen Hypervisor. http://www.xen.org.