NTNU
Innovation and Creativity

Permanent Storage Development for the J2EE Public Service at CERN

Ola Holmestad

Master of Science in Communication Technology
Submission date: March 2007
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Artur Wiecek, CERN (Artur.Wiecek@cern.ch)
Lucia Moreno Lopez, CERN (Lucia.Moreno.Lopez@cern.ch)

Norwegian University of Science and Technology
Department of Telematics

Problem Description

A current limitation of the J2EE public service at CERN is that no access to permanent storage is possible, only database access is given. The thesis will try to address this issue through the use of the WebDAV protocol, and it will be investigated how to integrate a WebDAV client into a running instance of a Tomcat container. One approach could be to intercept file requests and redirect these requests through WebDAV to another, remote, location.

The study will focus on security, transparency and integration with Tomcat for the users of the J2EE public service and ease of maintenance for the system managers. The following key elements will be investigated

- guaranteed confidentiality for the file system access
- transparency
- how it is possible to make this work when the application is moved to another container at the J2EE public service
- other security related elements

This will be followed by a development of an application in Java that fulfills these requirements and any other requirement discovered during the process.

Assignment given: 11. September 2006
Supervisor: Peter Herrmann, ITEM

Abstract

This Master thesis develops two solutions enabling permanent file storage for the J2EE Public Service at CERN by the use of Slide, a Java implementation of the IETF WebDAV protocol for distributed authoring and versioning.

Permanent storage development has been a requested feature for the J2EE Public Service since the startup in 2005, and in this thesis two libraries are investigated concerning how well they suit the J2EE Public Service environment, namely Jakarta Slide and Jakarta VFS. After research and test implementations, it is found that Slide performs well enough to be used in the future development of such a feature, while VFS does not because of adaption problems to the CERN environment.

On the basis of the research, two different solutions are implemented. The first solution is a stand-alone class that can be used for simple file/retrieval by providing specific methods for use by web application authors. After evaluation, this library is found to have several problems, especially concerning the missing transparency for the authors and no support for file hierarchy. The final solution is for this reason a re-implementation of the classes in Java that deals with file I/O, which provides a transparent and secure way of doing retrieval and storage from the J2EE Public Service.

Contents

List of Figures

List of Tables

List of Abbreviations

CERN	Conseil Européen pour la Recherche Nucléaire
HEP	High Energy Physics
LEP	Large Electron-Positron (collider)
SPS	Super Proton Synchroton
LHC	Large Hadron Collider
PHP	PHP: Hypertext Preprocessor
SSH	Secure SHell
J2EE	Java 2 Platform Enterprise Edition
WAR	Web ARchive
HTTP	HyperText Transfer Protocol
W3C	World Wide Web Consortium
WebDAV	Web-based Distributed Authoring and Versioning
RFC	Request For Comments
IETF	Internet Engineering TaskForce
MIME	Multipurpose Internet Mail Extension
URI	Uniform Resource Indicator
URL	Uniform Resource Locator
URN	Uniform Resource Name
HTML	HyperText Markup Language
TCP/IP	Transport Control Protocol/Internet Protocol
DNS	Domain Name System
SSL	Secure Sockets Layer
TSL	Transport Layer Security
RPC	Remote Procedure Call
XML	eXtensible Markup Language
OSI	Open Systems Interconnection
UTF	Unicode Transformation Format
ASF	Apache Software Foundation
VFS	Virtual File System
FTP	File Transfer Protocol
SFTP	Secure File Transfer Protocol
CIFS	Common Internet File System
DFS	Distributed File System

Preface

This Master thesis is written at the Department of Telematics of the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. The thesis work started in late September 2006 and ended in the beginning of March 2007, and was done at CERN, the European Organization for Nuclear Research, in Geneva, Switzerland.

Firstly, I would like to express my thanks to my supervisor at NTNU, Peter Herrmann, for accepting to supervise a thesis in a distant physics engineering environment such as CERN, and for his guidance during the thesis work.

In addition, I would like to thank all the members of the IT-DES-DIS section for continuous support and help during my stay at CERN. Special thanks goes to my initial supervisor for the Technical Student Program, Miguel Angel Marquina, and to the head of the IT-DES-DIS section, Eric Grancher, who both were very generous with advice and made my stay at CERN a pleasure.

I would also like to thank my two supervisors for this thesis, Artur Wiecek and Lucia Moreno Lopez, for all their guidance and suggestions during the work with the permanent storage project for the J2EE Public Service. I sincerely hope that they will have good use of the solutions developed in the future.

All errors and omissions in this thesis are the sole responsability of the author.

Introduction

This thesis came out from a growing need for the users of the J2EE Public Service at CERN to have a permanent storage solution that would enable them to get access to files stored at the central DFS file system at CERN from their web applications stored in the J2EE Public Service. Since this service was established in 2005, users have only had the possibility to use databases for saving and/or retrieval of information. In addition, there has always been possible to write to and read from a temporary directory, but the need for a solution that truly enables permanent storage for web application authors as easily as possible has been apparent.

Structure of the Thesis

I have chosen to divide this report into three different parts:

Part I is background theory that will support the next parts of this thesis. Chapter 1 gives a brief introduction to CERN and tries to explain CERN's mission, especially emphasizing today's most important projects. The Technical Student Program is presented, as well as the activities that are the most important in the DES group, especially in the DIS section where the author spent his time as a Technical Student.

The main service that this thesis concerns, namely the J2EE Public Service, is outlined in Chapter 2.

After this brief introduction, the HTTP & HTTPS protocols are presented in Chapter 3 as support for the WebDAV protocol, the main functional element of the thesis. The WebDAV protocol is explained in Chapter 4, and this concludes the first part of this report.

Part II concentrates on the research and the different test implementations that have been done. To introduce how file I/O in Java works, Chapter 5 is dedicated to this topic. Thereafter, two different library implementations, namely the Slide WebDAV library and the VFS library, both from the Jakarta project, are examined in Chapter 6.

For both these libraries, test implementations have been made, described in Chapter 7, to discover how well suited they were for this thesis' needs. The test implementation of the Slide Webdav Library is done entirely by the author, while the test implementation of VFS was provided with the VFS source code and adapted to the CERN environment only by minor changes. Lastly in this part, the need for further development and progress is summarized in Chapter 8.

Part III describes the two solutions that was developed by the author of this thesis for the users of the J2EE Public Service.

Chapter 9 explains the implementation of a stand-alone class File.java, while Chapter 10 gives an extensive description of how the final solution was designed, implemented and tested so as to provide the main functionality that was wanted before this thesis started.

A **Conclusion** ends the main part of this report.

Appendices

Appendix A gives an overview of the `java.io` *package, Appendix B shows how patching of the Slide WebDAV library to provide correct functionality was done, while Appendix C gives an overview of the structure of the electronic documentation provided together with this thesis.*

Part I

Analysis of background & context

Chapter 1

CERN... where the web was born!

CERN, the European Organization for Nuclear Research, is situated at the border between Switzerland and France, close to the city of Geneva and the Geneva Lake in Switzerland. The acronym CERN comes from the french "Conseil Européen pour la Recherche Nucléaire", which was a provisional body founded in 1952 to form a world-class fundamental physics research organization in Europe[1]. This body ordered in 1953 the construction of the worlds biggest particle physics center, which is what CERN is known as today.

After the second world war, America had a strong focus on research, and Europe needed to unite forces on this area. From this need, CERN was created to promote European Science. This chapter will give an insight into what CERN is, its major achievements in the past and current projects at CERN, both in the physics area and in the field of Information Technology.

1.1 Background & History

CERN was created in 1954 by 11 European countries[2]. At present, there are 20 member states that have both special duties and privileges, and they all contribute financially to the different CERN programs. This research laboratory is a highly attractive place to work for many of the world's best physicists and currently employs around 3 500 people[3], divided into physicists, engineers, technicians, scientific fellows, students, craftsmen, administrators, secretaries and workmen. In addition, somewhere around 6 500 visiting scientists from over 500 universities and over 80 nationalities are connected to CERN on a regular basis.

The mission of CERN is to develop tools, which means particle accelerators and detectors[2], that helps fundamental research in particle physics, and especially High Energy Physics (HEP). HEP is the study of the smallest particles, the building blocks, in our Universe. Particle accelerators accelerate particles up to velocities very close to the speed of light and then collides the particles at certain points where detectors are placed. These detectors make the particles visible and grants scientists the possibility of further analysis. High energy is needed to detect the

smallest particles we currently know of, any new particles that may appear in the collisions and to test new physics theories. Collisions and the detection of them all happens in an underground tunnel with a circumference of 27 kilometers where the LEP (Large Electron-Positron collider) was in operation from 1989 to 2000. Now, the same tunnel is used for the new LHC (Large Hadron Collider) which is being built and will be put in operation during 2007.

1.2 Achievements in the past

The research environment at CERN has led to many astonishing discoveries during the past, some of them of great importance for the way we live our lives today. This next section will give a short overview of some of these achievements.

1.2.1 Birth of the World Wide Web

In 1989, Tim Berners-Lee, a physicist at CERN, wrote a proposal on how to merge technologies of personal computers, networking and the concept of hypertext into a powerful and easy-to-use global information system, later known as the World Wide Web[4].

Berners-Lee and his companion Robert Cailliau, a systems engineer also at CERN, further developed the concept and wrote the formal proposal which was published on November 12, 1990. Within the end of 1990, Berners-Lee had built what was necessary for a working World Wide Web; the first web browser, called WorldWideWeb, the first web server (info.cern.ch) and the first web pages describing the project itself ([5], section History).

1.2.2 Achievements in physics

Nobel prizes

Three scientists at CERN have been awarded the Nobel prize in Physics[6].

In 1984, Carlo Rubbia and Simon Van der Meer were awarded the prize for their discoveries of the field particles W and Z, which confirmed the unification of the weak and the elecromagnetic forces.

In 1992, Georges Charpak was awarded the prestigious price for his inventions and development of particle detectors, especially for the construction of the multiwire proportional chamber in 1968, which marked the beginning of electronic particle detection. Technologies that are derived from these early achievements are now used in biological research and medical diagnostic tools.

Accelerator technology

In the 1950's, physicists realized that to make the most violent collisions, one had to fire two beams of particles at each other instead of firing a particle beam to a

stationary target as before. At CERN, a new approach was taken where a circular accelerator was built where two interconnected rings were feeded with particle beams and then made to collide. In 1971, the Intersecting Storage Rings (ISR) produced the world's first proton-proton collisions and CERN was after this considered a leader in the field of colliding beam projects[7]. Later, CERN developed the world's first proton-antiproton collider with the Super Proton Synchroton (SPS), and this led to the discovery of the W and Z field particles as described above

The LEP was CERN's next technical achievement, and now the LHC, shown in Figure 1.1, are built in the same tunnel as the LEP and are scheduled to run in 2007 to consolidate CERN's position in the world of accelerator technology.

1.3 The Large Hadron Collider

The LHC will be the largest and most powerful particle accelerator ever built, and with a cost for the accelerator of approximately 3 billion euros[8] also the most expensive one. It is under construction in the same tunnel that was used for the LEP accelerator, and it has taken a lot of collaboration between scientists and engineers to solve the different problems related to building such a big and complex structure.

In a particle accelerator, particles should be accelerated to the highest speed possible. In the LHC, the speed of the particles just before the collisions take place will be 0.999999991 the speed of light[9]. This means each particle will travel around the 27 kilometeres long circular tunnel 11 000 times pr. second, and collisions will occur 800 million times a second[10]. To bend the particles during their laps around the tunnel, 9 000 magnets will be used which in total create a magnetic field that is 200 000 times stronger than the earths own[11]. This could not have been accomplished without using the concept of superconductivity (the ability to conduct electricity without resistance or energy loss), and for this to happen the whole LHC needs to be heavily cooled. During operation, it will be cooled down to about 300 degrees below room temperature which will make it the coolest place in the Universe[12].

So what questions will scientists try to solve with the LHC and why is it necessary? Because our current understanding of the Universe is incomplete, the Standard Model that scientists now use still leaves unsolved questions, some of them are[13]:

- Why do elementary particles have mass, and why are their masses different?
- Why do the mass we can observe account for only 4% of the total amount of mass in the Universe?
- Why is the Universe made of matter and not antimatter?
- What happened during the first few milliseconds after the Big Bang?

The first question can be solved by the Higgs theory which involves a special particle called the "Higgs boson", which the LHC will be able to detect if it exists. In addition, the LHC will be able to confirm other physics theories, like a possible unification of the four forces and the concept of supersymmetry (the theory that for

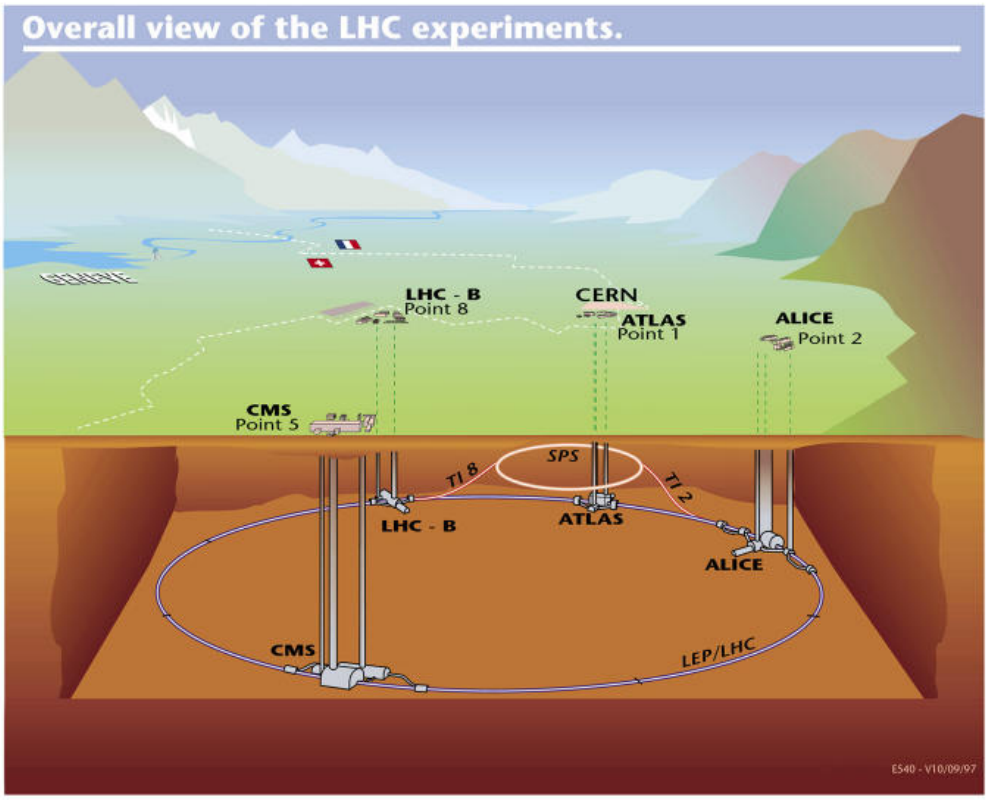

Figure 1.1: Illustration of the LHC and the four different detectors

each known particle there exists a "supersymmetric" partner. LHC will also be able to detect these if they exist[13]).

1.4 Important IT projects at CERN

The LHC will generate huge amounts of data, which create a need for a lot of data treatment and analysis. There exists a number of IT projects at CERN closely connected to the LHC, and some of them will briefly be explained below.

1.4.1 LHC Computing Grid

The LHC particle accelerator will generate 15 PetaBytes (15 million Gigabytes) of data annually[14]. It was clear from the beginning of the planning of LHC that new ways to deal with this enormous amount of data had to be found. The LHC Computing Grid (LCG) has as its mission to build and maintain a data storage and analysis infrastructure for the LHC, which includes the entire high energy physics community that will use the LHC.

The LCG will be based on a four-tiered model, with distribution of data around the globe[15]:

- Tier-0 center of LCG is tape recording of the raw data at CERN

- Tier-1 centers will lie around the world and will receive initially processed data from the LHC
- Tier-2 centers will provide computing power for specific tasks
- Tier-3 will typically be individual scientist's computer resources or university computer resources that individual scientists can use

Running the LHC means that over 5000 scientists from more than 500 research institutes and universities worldwide need access to the data created, and this data needs to be available during the 15 years estimated lifetime of the LHC. Analyzing these data and comparing with theoretical simulations require approximately 100 000 CPU's at 2004 measurements ([15], *section "Rationale for the LCG project"*).

The use of a computing grid was chosen because ([15], *section "Rationale for the LCG project"*):

- The enormous costs are more easily handled in distributed environments
- There are no single points of failure

There are also several challenges that need to be overcome; some of these are network bandwidth, the problem of different software versions, heterogenous hardware and the management and protection of the data.

1.4.2 EGEE

The Enabling Grids for E-SciencE project[16] is closely connected to the LCG. It is a cooperation between 90 institutions in 32 countries worldwide, and aims at developing a seamless Grid infrastructure for scientists that is available 24 hours a day. When time and resources get too big, it becomes impractical to use traditional IT infrastructures for such tasks. This grid consists of 20 000 CPU's, 5 Petabytes of storage, can handle over 20 000 concurrent jobs, and is distributed across 200 sites in 39 countries.

1.4.3 CERN OpenLab

OpenLab[17] is a collaboration between CERN and industrial partners, specially aimed at developing data-intensive solutions for the LHC. It gives a framework for evaluating and integrating cutting-edge technologies and services from the industry leaders. At the same time, these industry leaders can carry out large-scale highly performance evaluation of their solutions in an advanced research environment.

Partners for the CERN OpenLab project are currently HP, Intel Corporation and Oracle, while F-Secure and StoneSoft also contribute to the project.

1.5 The Technical Student Program

The Technical Student Program[18] at CERN is aimed at undergraduate students in technical fields and gives them a chance to work on a project in a research environment for six to twelve months. The students can from this work gain important experience and at the same time give valuable contribution to the work in various groups and fields at CERN.

For my part, I was a technical student in the DES group and the DIS section, and my tasks during my 11-month stay at CERN included but were not limited to the following:

- General Oracle database account administration
- Further development of Java application dealing with security scans of the most important databases at CERN
- Development of a PHP web interface that handles access to computers at CERN through the use of SSH keys
- This thesis work, creating a permanent storage solution in Java for the J2EE Public Service

1.6 The DES group

The Database and Engineering Services group at CERN is part of the IT Department.

Main activities include[19]:

- Central database services
- Support of engineering software packages, like tools for Mechanical, Electronic and Software Design
- CVS support, software license management, J2EE Public Service and system support for TWiki

1.6.1 The DES-DIS section

The Database Infrastructure Services section is where I was working during my stay at CERN. Its main activites include[20]:

- Central database support for services based on the Oracle Database & Application Server
- Installation, distribution, consultancy, first-line support, database management and administration of all of the main CERN databases
- Supplier relations and license/contract management (mainly Oracle)

Chapter 2

J2EE Public Service

The J2EE Public Service is a central server infrastructure for deployment of Java server-side, namely servlet/jsp, applications[21]. This service provides a place to deploy and run web applications for CERN users, and it is run within the IT-DES group.

2.1 Background

After a request from the Java community at CERN, it was decided that a robust centrally managed deployment platform for Java server-side applications should be implemented. The goal was to find a solution that enabled load balancing, easy administration and a high level of isolation between different applications ([22], *Abstract*). To ensure a coherent view of web deployment platforms at CERN, the service should also be closely integrated with the CERN Web Services run by the IT-IS group.

This need came from a big increase of users running web applications on their own computers. To do this, one has to install an application server (like Apache Tomcat, Oracle OC4J or Oracle iAS) on the local computer, and this opens up great security risks when it comes to enforcing new security patches. It also requires a lot of skills and time to make it work correctly. Other reasons to migrate to a centrally managed infrastructure were the ability to have reliable hardware, hardware redundancy, backup and monitoring of the application servers ([22], *Introduction*).

At the 1st of February 2007, the J2EE Public Service is storing 151 applications in 103 containers belonging to 48 different users[23]. The application server that currently is being used is Apache Tomcat 5.5.15 on Java 2 SDK version 1.5.0.

The service targets medium-sized and relatively important applications, but not mission-critical. In addition, it supports deployment, not development, which means that users will get no help with their source code and need to make sure that their applications work correctly before they deploy them. The service provides[24]:

- Scalable server-side infrastructure
- A way to deploy applications

- Tools for monitoring of the applications
- Mechanism to avoid excessive use of resources
- Procedures for easy reinstallation
- Backup and recovery procedures
- User documentation

2.2 How does it work?

The J2EE Public Service is implemented with the use of a multiple container approach, as shown in Figure 2.1. Each machine hosts multiple containers, i.e. instances of Tomcat. Each web author uses one container, but has the possibility of managing several applications in this container[25].

It is possible, though not recommended, to host applications from different web authors in the same container. If this is done, there is a possibility that the applications might interfere with each other and not work as they are supposed to. In addition, the security is implemented on container level, which means that to do this would imply a possible security breach.

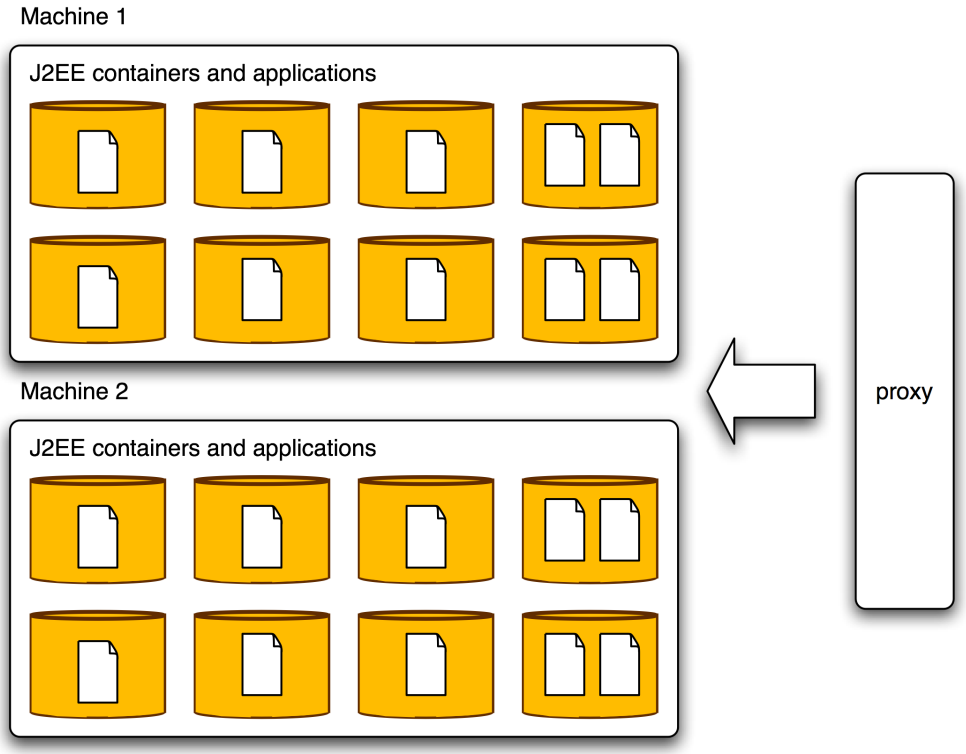

Figure 2.1: The multiple container approach

The containers all listen to a set of different ports that are reserved exclusively for each container. At the same time, the web readers, which are the actual users of the

applications, connect using the standard HTTP (port 80) and HTTPS (port 443) ports. To make this work, a proxy is used to connect the requests from web readers to the correct ports of the container.

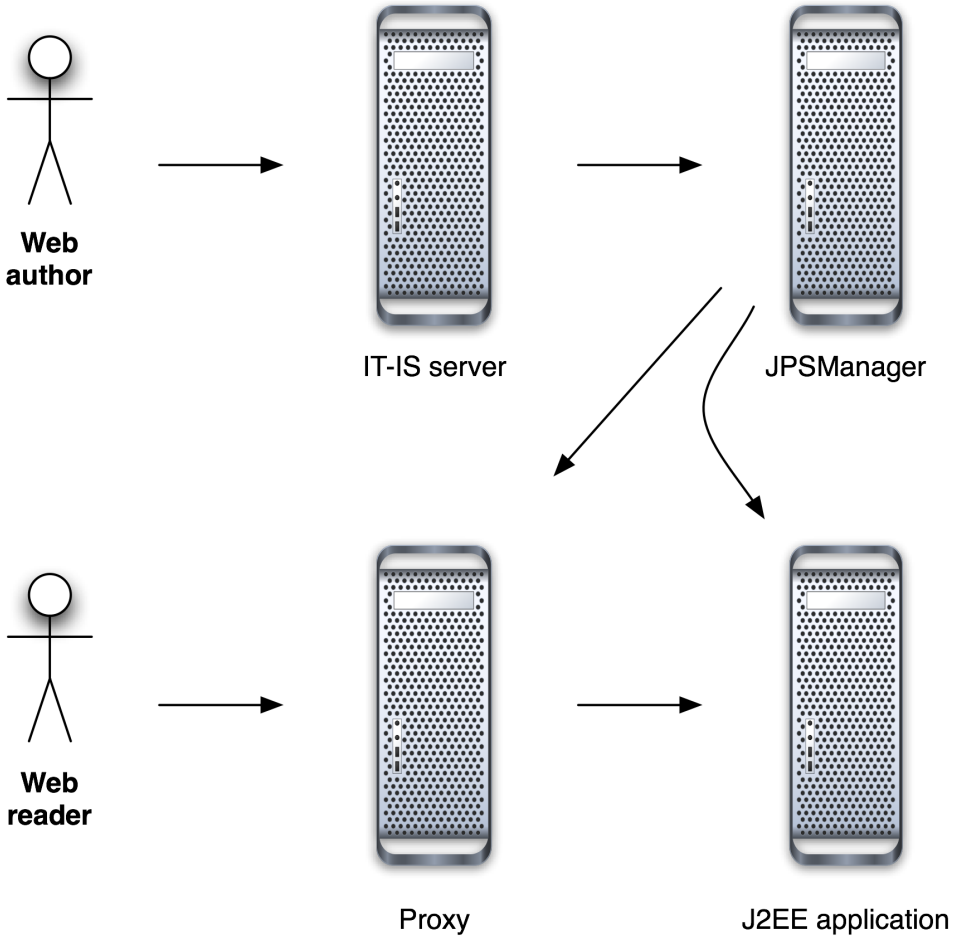

Figure 2.2: How does the J2EE Public Service work

When a web author wants to deploy an application to the J2EE Public Service, the web author has to do the following, illustrated in Figure 2.2:

- Register a new application with the CERN Web Services run by the IT-IS group. This request will typically be forwarded to JPSManager
- JPSManager will then create a new container on one of the application hosts and update the proxy so that it can forward requests correctly
- The web author should then deploy the application through HTTPS via a web interface

2.2.1 WAR-file

The format of the application files that users should upload to the service is that of a Web ARchive file, as shown in Figure 2.3:

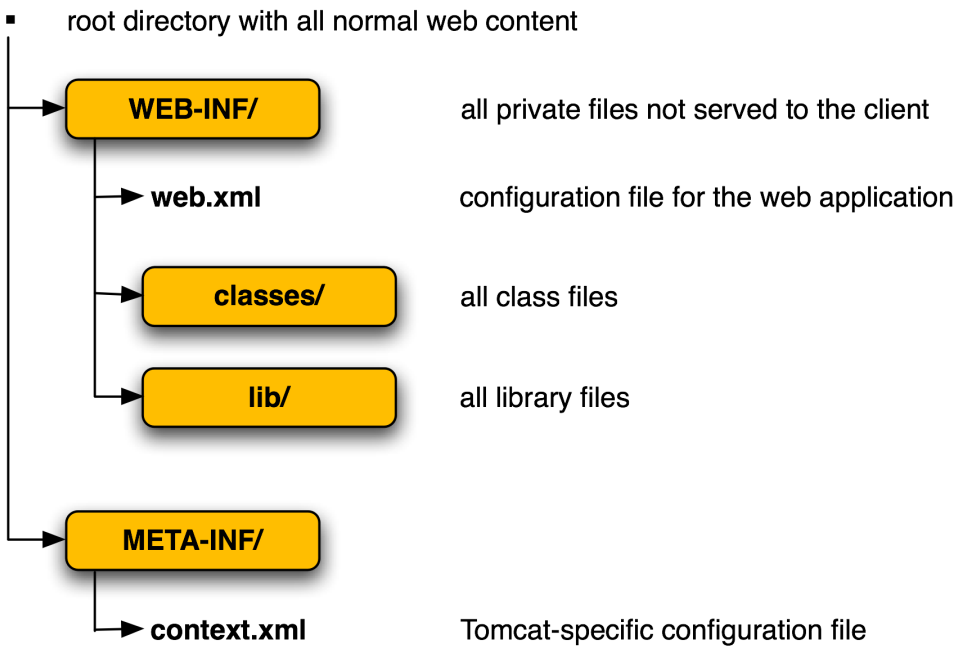

Figure 2.3: Structure of a WAR file

2.3 Architecture

The J2EE Public Service consists of a distributed environment without any shared storage. There is no redundancy when it comes to hardware, but the architecture itself makes the recovery procedures simple[26].

The architecture consists of a cluster of 5 HP machines, as shown in Figure 2.4. All the machines have the same configuration making the architecture easily scalable.

The communication between the different nodes is implemented by the use of SSH version 2 for start/stop of containers remotely, rsync over SSH version 2 for synchronization of disk files and HTTP/HTTPS/AJP13 for the communication between the Apache proxy and the Tomcat containers

Security for the service is taken care of by the use of different operating system accounts, which enables file system access rights. In addition, the Java security manager is used to give fine-grained control about which operations each application has the privileges to perform.

2.4 JPSManager

JPSManager is a CERN-developed solution in Java for creation and administration of application servers. It enables deployment of a web author's web application by configuring a separate web container for each new user on request from the CERN Web Services.

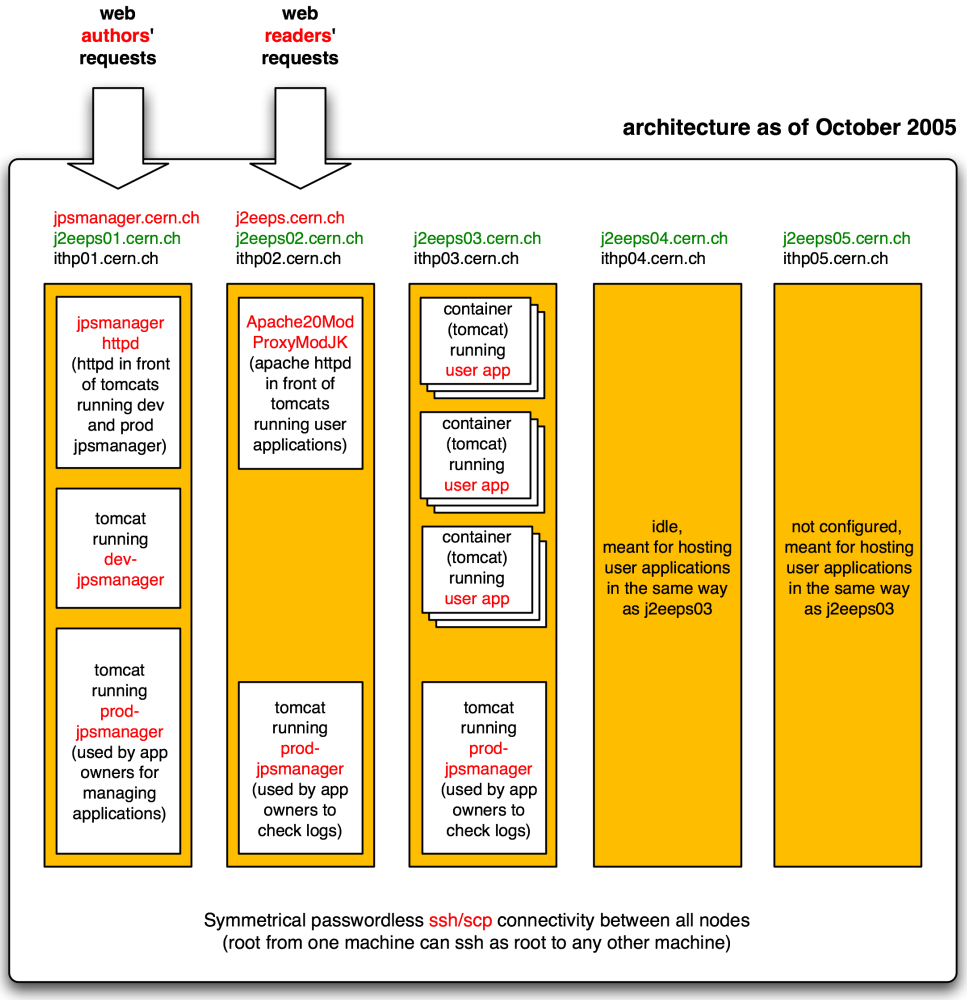

Figure 2.4: J2EE Public Service architecture

For administration, there exists a web interface deployed on Tomcat and a command-line interface.

2.4.1 Architecture

The architecture of the JPSManager[27] is centered around 3 main interfaces, as shown in Figure 2.5:

- JPSContainerManager
- JPSProxyManager
- JPSContainerAssigner

JPSContainerManager defines methods that has to be implemented when a new type of container is to be used. JPSProxyManager defines methods for the use of a

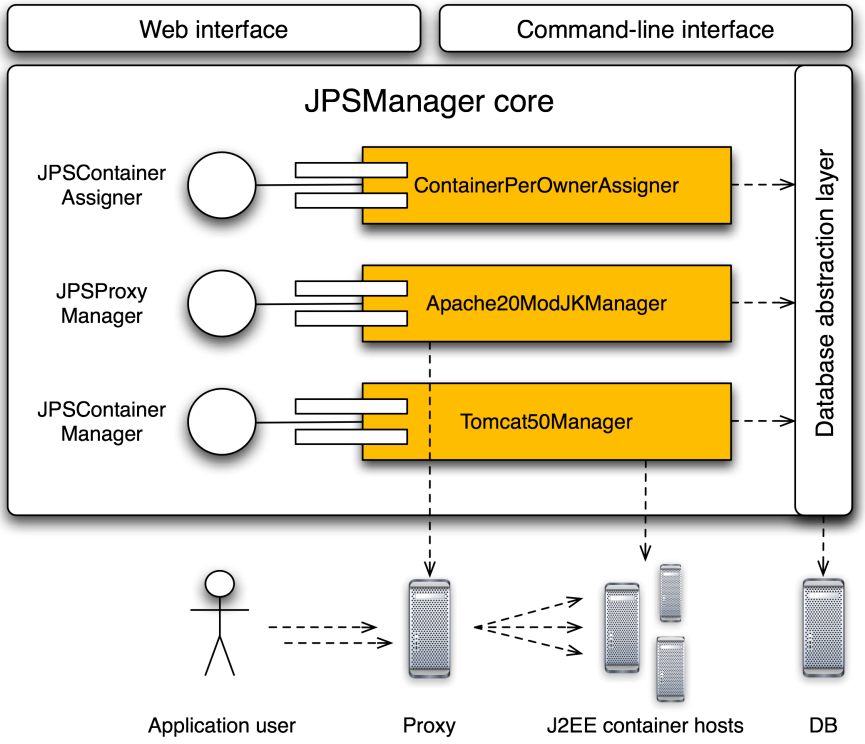

Figure 2.5: Architecture of JPSManager

specific type of proxy, while JPSContainerAssigner defines policies on how to assign containers to web authors.

2.5 Permanent storage

From the beginning, users had no possibility of storing or retrieving files. The temporary directory of each container has been available to write/read files, but the fact that users do not have the possibility to put files in this directory or retrieve files from the directory is a limitation to the service that many users have been asking for. This has been one of the elements that the administrators of the J2EE Public Service has wanted a solution for, and this thesis provides this.

Chapter 3

The HTTP & HTTPS protocol

This chapter will provide an overview of the HTTP protocol and the secure version of it, HTTPS. The goal is to give the necessary information to understand the next chapter about the WebDAV protocol.

3.1 HTTP

The HyperText Transfer Protocol is used for information transfer across the World Wide Web. It is defined as an Internet specification[28] by the IETF.

3.1.1 Background

The HTTP protocol has been in use since 1990. The first version was called HTTP/0.9 and was basically a simple protocol for the transfer of raw data, originally meant for retrieval and publishing of HTML pages. The second version, HTTP/1.0 came in May 1996 and was an improvement of the 0.9 version by allowing a MIME message format for the message bodies and persistent connections ([29], *HTTP versions*). Today's version, HTTP/1.1, is the final version of HTTP and further work has been closed by the W3C[30] because the standard is successful and this latest version addresses the weaknesses of the earlier versions.

HTTP is an application-level request/response protocol and the message exchange is most often initiated by a user agent or a client requesting a specific resource on a server. The protocol builds on the principle of reference provided by the URI[31] scheme as a URL or URN. It normally works by using the underlying TCP/IP ([32], [33]) transport protocol, by setting up such a connection to a specific port (port 80 by default) on a remote host. Subsequently, it can be implemented on top of any other transport protocol on other networks than the Internet. All the protocol needs is a reliable transport mechanism, which most commonly is TCP/IP today.

3.1.2 HTTP messages

An HTTP message is either a request or a response message. HTTP is based on a client-server model, where clients request information and servers respond with the requested information or with an appropriate error message if the request could not be treated correctly.

In general, both types of messages consist of ([28], *Chapter 4.1*):

- A start line
- Zero or more header fields
- An empty line to inform that the header is finished
- An optional message body that depends on the nature of the request/response

The header fields of the two types of messages are further divided into four groups that each logically links to the messages or the different parts of the messages ([28], *Chapter 4.2*):

- General header fields, which are common for both types of messages
- Request-headers, which gives the client an opportunity to describe itself and additional information about the request
- Response-headers, which gives the server an opportunity to describe additional information about the response
- Entity-headers, that define metainformation about the message body or about the resource identified in the request

The format of the header fields is the name of the header followed by a colon and a value.

The message body is used for additional data information and conveys for example a part of an HTML document. If a message body is included in the message, either a `Content-Length` or a `Transfer-Encoding` header field has to be included in the header of the message.

Request

In a request message the first line includes the method that should be used for the given resource specified by the `Request-URI` in the message. In addition, an identifier of the resource and the protocol version that is being used has to be specified ([28], *Chapter 5*), as shown in Figure 3.1.

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Figure 3.1: Simple example of a HTTP request message

Response

In a response message, the first line is a status line where the protocol version, a numeric status code and the status code's textual explanation is stated ([28], *Chapter 6.1*). Figure 3.2 shows an example of a response message with some header fields included (message body omitted).

```
HTTP/1.1 200 OK
Date: Wed, 07 Feb 2007 15:29:35 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-modified: Wed, 10 Jan 2007, 23:12:12 GMT
Content-Length: 567
Content-Type: text/html; charset=UTF-8
```

Figure 3.2: Example of a HTTP response message

3.1.3 Status codes

Table 3.1 shows the different status code classes that are used by HTTP ([28], *Chapter 10*)

Status code	Textual description	Explanation
1xx	Information	Provisional response
2xx	Success	Received, understood and accepted message
3xx	Redirection	Further action needs to be taken by the the client before request can be considered successful
4xx	Client error	Client performed an error
5xx	Server error	Server is not capable of performing the request

Table 3.1: Overview of HTTP status codes

Table 3.2 shows some of the most used HTTP status codes ([28], *Chapter 10*)

3.1.4 HTTP methods

This part will briefly describe the different methods defined in HTTP/1.1 ([28], *Chapter 9*)

Status code	Textual description	Explanation
100	Continue	The client should continue with his request
200	OK	Successful request
301	Moved Permanently	The requested resource has been moved permanently to a new location
400	Bad Request	Request not understood (bad syntax)
403	Forbidden	Request understood, but server refuses to fulfill it
404	Not Found	The resource was not found on the server
500	Internal Server error	The server encountered an internal error and cannot fulfill the request
503	Service Unavailable	The server was not able to handle the request at the current time because it might be overloaded or performing maintenance operations

Table 3.2: Some widely used HTTP status codes and their explanations

Safe methods

The safe methods defined in HTTP/1.1 are the GET and HEAD methods ([28], *Chapter 9.1.1*). For a method to be safe, there cannot be any other action than a pure retrieval resulting from the method. It is not possible to guarantee this, but users should not be responsible for undesired actions at the server resulting from such a method.

OPTIONS

The OPTIONS method ([28], *Chapter 9.2*) is used for information requests about which communication options are available for the request/response message exchange. Use of this method can give information about requirements enforced by the server and server capabilities, without actually performing any operation on the server.

GET

The GET method ([28], *Chapter 9.3*) is used to retrieve information from the resource identified by the `Request-URI` in the request.

HEAD

The HEAD method ([28], *Chapter 9.4*) is identical to the GET method, but there is no message body returned in the response. This method is used to obtain information about the resource identified by the `Request-URI` without retrieving the actual

entity, and is often used for testing links for validity, accessibility and when they were last modified.

POST

The POST method ([28], *Chapter 9.5*) is used to add a new entity to the resource identified by the `Request-URI` in the request message. This method is used for:

- Message posting on discussion forums, guestbooks and other similar web applications
- providing data to a data-handling process
- providing new data or change existing data in a database
- adding of information to an already existing resource

PUT

The PUT method ([28], *Chapter 9.6*) is used to store the entity of the message body in the request message under the `Request-URI` from the request message.

The main difference between PUT and POST is that the `Request-URI` in a POST method request identifies the source that should handle the entity in the message body, while the `Request-URI` in a PUT method request identifies the entity itself in the message body.

DELETE

The DELETE method ([28], *Chapter 9.7*) requests that the resource identified by the `Request-URI` in the request message should be deleted. Even if the status code in the response message to the message that issued a DELETE method request indicates success, there is no guarantee that the deletion was performed. However, the server should never send a success response message if it does not think it is able to delete or relocate the resource to an inaccessible destination.

TRACE

The TRACE method ([28], *Chapter 9.8*) performs a loop-back of the request message. It is useful for testing to see whether what the server received was the same as what the client sent and use the response for diagnostics if this was not the case.

CONNECT

The CONNECT method ([28], *Chapter 9.9*) is reserved by the specification for the use of a proxy that can switch dynamically to using SSL tunneling.

3.1.5 Persistent connections

Before, one TCP connection was established each time a URL was to be fetched using HTTP. This increases load on HTTP servers and can cause congestion on the Internet during heavy-load periods. With the use of persistent connections, introduced in HTTP/1.0, it is allowed to use one TCP connection for multiple requests. This approach has numerous advantages ([28], *Chapter 8.1.1*):

- Fewer TCP connections are opened which means that CPU time are saved all over the internet in routers, clients and hosts
- HTTP requests can be pipelined over one TCP connection, this means one can send a new request without waiting for the response of the last request. Again, more efficient use of the TCP connection
- A smaller number of packets for establishing TCP connections hereby reducing network congestion
- The latency of message sessions is reduced because only one TCP handshake procedure is done for multiple requests
- HTTP can evolve easier, new features can be tried out without risking to close the TCP connection

As a difference from earlier HTTP versions (namely HTTP/1.0), persistent connections is the default behavior in HTTP/1.1 ([28], *Chapter 8.1.2*)

Practical considerations

The use of persistent connections in HTTP/1.1 have some practical usage implications ([28], *Chapter 8.1.4*). Normally, most servers will have a time-out value where they will no longer keep a connection open. In HTTP/1.1, there is no requirements on this time-out value. This means that clients and servers should close the connection when they are finished, instead of relying on any time-out value from the server.

Both clients and servers must also be able to recover from asynchronous close events, i.e. if the server sent a close message before receiving a new request from the client and the client sends new requests and then receives a close message from the server.

In addition, good practice for clients is to limit the number of connections to the server. The recommendation is for a client not to have more than two connections to a proxy or a server.

3.1.6 Security considerations

There are some security limitations in HTTP/1.1 ([28], *Chapter 15*). First of all, a HTTP client often have control of much personal and privacy information, like username, mail address and passwords. Leakage of such information from the use of the HTTP protocol should be prevented.

Secondly, server logs containing information about users requests should not be abused or distributed in any way.

Thirdly, HTTP offers no content regulation, so transfer of sensitive information has to be kept secure by the applications using HTTP.

In addition, file restrictions and restrictions on path names should be strictly enforced and files that are not meant to be viewed from the outside world (like access control files, configuration files and script code) should be correctly protected.

As a last point, clients should be careful about using IP adresses from previous sessions, because they might change at any time. Instead, DNS should be used for resolving IP addresses rather than caching of old DNS entries.

3.2 HTTPS

Currently there are two methods of implementing security to HTTP, the HTTPS URI scheme and the HTTP/1.1 Upgrade header ([29], *Section "Secure HTTP"*). The first option has actually been deprecated, but is still widely used for establishing secure connections, while the HTTP/1.1 Upgrade header is not much supported yet.

3.2.1 HTTPS URI scheme

The HTTPS URI scheme works by adding an encryption layer of SSL/TLS to protect the data being sent. It is syntactically identical to HTTP, but uses port 443 instead of port 80 for communication.

3.2.2 HTTP/1.1 upgrade header

The `upgrade` header was introduced in HTTP/1.1. As the name implies, it is used to upgrade an HTTP session from normal, clear-text communication into a secure communication channel using TLS.

3.2.3 SSL

Secure Sockets Layer (SSL) was introduced by Netscape Communications Corp. in 1995 and is now widely used. It provides ([34], *page 813*):

- Negotiation of security parameters between a client and a server
- Mutual authentication of client and server
- Secret communication
- Protection of data integrity

OSI protocol overview
Application (HTTP)
Security (SSL/TLS)
Transport (TCP)
Network (IP)
Data link (PPP)
Physical

Table 3.3: The OSI Model protocol stack with SSL/TSL shown

SSL is a new layer positioned between the application layer, which HTTP is part of, and the transport layer, in the OSI Model protocol stack. Table 3.3 shows a typical Internet example of the protocol stack ([34], *page 814*).

SSL consists of two subprotocols ([34], *page 814*). One handles the establishment of a secure connection while the other handles the actual use of the secure channel.

TSL

Transport Layer Security (TSL[35]) is a standardized version of SSL. Compared to SSL, the changes done are minor, but they are big enough that interoperability between SSL and TSL does not exist.

Chapter 4

The WebDAV protocol

Web-based Distributed Authoring and Versioning (WebDAV) is a set of extensions to the HTTP protocol that was first proposed as an Internet Standard in February 1999[36]. It allows users to edit and manage files on remote web servers. This chapter aims at explaining the key features of WebDAV and give a brief description of the different methods that are available for use.

4.1 Background

The WebDAV protocol provides a set of methods, headers and content-types for the management of resource properties, creation and management of resource collections, namespace manipulation and resource locking ([36], *Abstract*). A resource is web terminology for any piece of information, such as a web page, a document or an image, and its location is described by some sort of URI. WebDAV provides operations for remote web content authoring like ([36], *Chapter 1: Introduction*):

- Properties: Create, remove and query information about Web pages such as author and creation date
- Collections: Create documents and retrieve files/directories from an hierarchical system
- Locking: Keeping more than one person from working on a document simultaneously
- Namespace operations: Copying and moving Web resources

The benefits of WebDAV are[37]:

- It gives a possibility of content publishing to the web through HTTP
- It provides collaborative authoring with the use of locking to prevent overwrite conflicts
- There are no document format restrictions

- There is a common interface for a wide range of repositories

WebDAV chose to add methods to the existing HTTP protocol to provide a standard for a writable and collaborative web. In this process, several options were investigated. These options are URL munging, RPC-via-POST and the adding of methods([38], *Chapter "How WebDAV extended HTTP"*), and they will be further discussed below.

4.1.1 URL munging

URL Munging describes how to append commands after the end of a URL, typically after a "?" sign, much like the widely used script language PHP does today. This has the advantage of being easily parsed, but the main problem with this method is that only the HTTP GET method can be used for retrieval of Web contents.

4.1.2 RPC-via-POST

A possibility of injection of parameters into the message body along with a function identifier and then the use of the HTTP POST method was also considered for the WebDAV specification. The server would then perform the necessary operations based on the function identifier and the parameters included and return the answer in the message body of the response message.

The main drawback of this way was that it would leave the POST method a security hole, since virtually any operation then could have been done through the POST method.

4.1.3 Adding methods

The adding of methods to the already existing methods of HTTP takes advantage of existing HTTP features, hereby providing security and easy access control. The main drawbacks of this method was that since data can have unbound length, it is not always suitable to encode it in a header like in existing HTTP. New methods would also have to be integrated with the old HTTP headers.

The advantages of the adding methods approach when it comes to security and access control issues were so big that this solution was obvious for the people working with the WebDAV draft. In addition, they also chose to use some elements from the RPC-via-POST solution by the use of XML in the body of the messages. This means that WebDAV encodes method parameter information either in an XML request body or in HTTP headers.

4.2 Properties

Properties ([36], *Chapter 4.1*) are data about data (metadata) and in this context they describe the state of a certain resource. They are name/value pairs, the name

is unique for the resource, and existed already in some sense in HTTP as HTTP message headers.

Properties for a resource are split into “live” and “dead” properties. A “live” property is controlled by the server, and it can either be the server that fully controls the value of the property or the client that maintains the property while the server controls the syntax of the submitted values. A “dead” property is controlled and set by the client, and the server only stores the value of the property given from the client.

The state of a resource can possibly be described by a large number of properties and it is clearly inefficient to always use them all. Because of this, there was a need for a mechanism that allows identification of a set of properties and how to set them ([36], *Chapter 4.3*) , and this is done in the WebDAV specification using the PROPFIND method described in Chapter 4.5.1 below.

4.3 Collections

Collections ([36], *Chapter 5.2*) in WebDAV are seen much as file system directories within a server's namespace. Each collection has a list of internal member URI's and a set of properties connected to it.

4.4 Locking

Locking ([36], *Chapter 6*) is the concept of serializing access to a resource such that no modification is possible while another party is editing the resource, otherwise the update done by one or more of the parts could be lost. For a server to comply with the WebDAV standard it does not necessarily need to support locking. Different storage repositories need to control their own storage management, if locking should be available, and what sort of locking should be used ([36], *Chapter 6.2*). For clients, they can either try to issue a lock and see if that works or try to investigate the locking capabilities of a server through the use of lock capability discovery.

4.4.1 Exclusive locks

An exclusive lock ([36], *Chapter 6.1*) means that only one single principal can access a resource, and this guarantees that the edits done by this source will work with no overwrite conflicts.

4.4.2 Shared locks

Shared locks ([36], *Chapter 6.1*) can be used when several principals need access to a resource. These principals will then have the permission to issue a lock on a specified resource. This approach trusts that other principals will not overwrite uncommitted changes and implies letting principals that can receive a lock know who else may be working on the resource for the moment.

4.4.3 Usage considerations

Even though locking is used, one cannot guarantee that updates do not get lost. This is because the WebDAV standard also needs to correctly give access to clients that are only HTTP compliant ([36], *Chapter 6.7*).

Consider a scenario where two clients, A and B, where A is a HTTP client that is not able to perform locking and B is a WebDAV compliant client. First, A does a HTTP GET on a resource and starts editing. In the meantime, B locks the same resource, does a HTTP GET, edits the resource, issues a PUT and unlocks the resource. Then, when A is finished with editing and uses PUT to save it back to the server, all B's changes are lost.

Since not all clients can be forced to use locking because WebDAV has to be HTTP compatible, this cannot be prevented. In addition, it cannot be required that servers support WebDAV, and since WebDAV is stateless, there is no possibility to enforce a series of operations like LOCK - GET - PUT - UNLOCK.

4.5 HTTP methods

Below is an overview of the different methods that WebDAV provides ([36], *Chapter 8*) as an extension to the HTTP methods described in Chapter 3.1.4

4.5.1 PROPFIND

The PROPFIND method ([36], *Chapter 8.1*) retrieves properties defined on the resource identified by the `Request-URI` in the request message, and also on member resources if any are present. The `Depth` header can be used to specify to which children level the method should return results, either "0", "1" or the default "infinity" can be used as values.

Using the body of the request message, clients can request either specific property values, all property values available or a list of the property values available, where default behaviour if nothing is specified is to retrieve all property names and values.

4.5.2 PROPPATCH

The PROPPATCH method ([36], *Chapter 8.2*) is used to set or remove properties on the resource identified by the `Request-URI` in the request message.

4.5.3 MKCOL

The MKCOL method ([36], *Chapter 8.3*) is used to create a new collection (directory) on the resource identified by the `Request-URI` in the request message. For this method to succeed, all subdirectories must already exist. A message body may also be included when this method is used, specifying members of this new collection

and their properties. An example of a request message using the MKCOL method is shown in Figures 4.1 and 4.2

```
MKCOL /test HTTP/1.1
Host: www.example.com
```

Figure 4.1: Request message using MKCOL

```
HTTP/1.1 201 Created
```

Figure 4.2: Response message to the above request

4.5.4 DELETE

The DELETE method ([36], *Chapter 8.6*) deletes either the non-collection resource (a file) specified by the `Request-URI` in the request message or the collection specified by the `Request-URI` and all children of this collection. An "infinity" value of the `Depth` header is always used. Figures 4.3 and 4.4 shows an example where the request to delete the directory www.example.com/container failed because the internal member resource3 was locked.

```
DELETE /container/ HTTP/1.1
Host: www.example.com
```

Figure 4.3: Request message using DELETE

4.5.5 PUT

For a non-collection resource, the PUT method ([36], *Chapter 8.7*) adds (or replaces if the resource already exists) the resource specified by the `Request-URI` in the request message and changes values of the properties of the resource.

For collections, the use of the PUT method is not defined in the WebDAV specification, instead the MKCOL method should be used.

4.5.6 COPY

The COPY method ([36], *Chapter 8.8*) copies the resource from the `Request-URI` specified in the request message to the location specified in the URI in the `Destination` header field, which always has to be present in the message header for this method to work.

For collections, all internal members should also be copied to the destination, as the value of the `Depth` header field is always assumed to "infinity" if nothing else is specified.

If a resource already exists at the destination, the actions taken depends on the settings of the `Overwrite` header field. If this header field is set to "T" then a

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
   <d:response>
      <d:href>http://www.example.com/container/resource3</d:href>
      <d:status>HTTP/1.1 423 Locked</d:status>
   </d:response>
</d:multistatus>
```

Figure 4.4: Response message to the above request

delete with depth "infinity" should be performed, if this header field is set to "F" the operation will fail. The default value of the `Overwrite` header is "T" if it is not set.

Figures 4.5 and 4.6 shows an example of use of the COPY method where the request asks for the collection `www.example.com/container/` to be copied to `www.example.com/othercontainer` while maintaining all the live properties of the collection and its internal members. From the response message in Figure 4.6 it can be seen that most of the resources in the collection was successfully copied to the new location. However, the collection R2 failed most likely because of a problem with copying of some of the live properties.

```
COPY /container/ HTTP/1.1
Host: www.example.com
Destination: http://www.example.com/othercontainer
Depth: infinity
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<d:propertybehavior xmlns:d="DAV:">
   <d:keepalive>*</d:keepalive>
</d:propertybehavior>
```

Figure 4.5: Request message using COPY

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
   <d:response>
      <d:href>http://www.example.com/othercontainer/R2/</d:href>
      <d:status>HTTP/1.1 412 Precondition failed</d:status>
   </d:response>
</d:multistatus>
```

Figure 4.6: Response message to the above request for the COPY method

4.5.7 MOVE

The MOVE method ([36], *Chapter 8.9*) performs the same operation as the COPY method, but in addition it also deletes the source collection with all internal members or the file specified in the `Request-URI` of the request message. It also does maintenance processing, updating the URI's referring to the old resource if this is necessary.

This method works only with the `Depth` header set to "infinity", and the behavior concerning the `Overwrite` header field is the same as for the COPY method.

4.5.8 LOCK

The LOCK method ([36], *Chapter 8.10*) is used to issue a lock on the resource specified by the `Request-URI` in the request message. The scope of the lock is the entire state of the specified resource, this includes all properties and for collections the ability to remove or add internal members.

The `Depth` header field can be used, with values of either "0" or "infinity", where "0" means only the resource itself while "infinity" means all internal members to the end of the hierarchy.

4.5.9 UNLOCK

The UNLOCK method ([36], *Chapter 8.11*) removes the lock from the resource specified by the `Request-URI` in the request message.

4.6 Security considerations

All of the risks described in Chapter 3.1.6 are also valid for WebDAV. In addition, since WebDAV creates more possibilities for remote authoring, some new security threats arise ([36], *Chapter 17*):

4.6.1 Authentication

The use of authentication technologies are necessary to protect access and integrity of resources. Basic HTTP authentication is not sufficient, because password is sent in clear-text. Instead, a secure connection like HTTPS using SSL/TSL as described in Chapter 3.2 should be used, or the network should be physically secure ([36], *Chapter 17.1*).

4.6.2 Denial of Service

Denial of Service attacks[39] aims at making a computer resource unavailable for its users. The use of WebDAV plus HTTP possibly enables such attacks on every

part of a system's resources. These attacks can be using the PUT method to store large files, asking for recursive operations which take up a lot of processing time or making pipelined requests on multiple connections to cause network connection congestion ([36], *Chapter 17.2*).

4.6.3 Security through obscurity

The PROPFIND method actually provides a method to list properties for a given resource that an attacker would have to search for otherwise. It is therefore strongly advisable that access control techniques are used, the security should not depend on the obscurity of resource names ([36], *Chapter 17.3*).

4.6.4 Privacy concerning properties

Properties can contain privacy information, such as author names. Again, access control mechanisms should be used to prevent accidental read of properties. A solution with separation of read access to the resource body and the resource properties could also be used ([36], *Chapter 17.5*).

4.6.5 Source files

WebDAV can potentially give access to source files containing sensitive information. Caution must be made to prevent unauthorized principals getting read/write access to such files. ([36], *Chapter 17.6*)

Part II

Research & Test Implementations

Chapter 5

Java I/O

5.1 General

Java I/O is a library that provides system input and output through the use of data streams, serialization and the local file system. The basic principle for Java I/O is that each class in the package is designed for one special task and must be combined with other classes to perform more complex tasks. Layering of multiple objects is used to provide the desired functionality. For instance, a `FileReader` provides a way to connect to a file, and nothing else. A `BufferedReader` provides buffering of input. Used together, these two classes provide buffered reading from a file ([40], *Section "Basic Principles"*). This chapter will give an introduction to I/O in Java, focusing on file I/O.

There are two kinds of I/O, namely byte and character I/O ([40], *Section "Introduction"*).

5.1.1 Byte oriented I/O

Byte oriented I/O is meant for data further processed by computers and not by humans. The classes mainly used for this type of I/O are the InputStream and OutputStream classes in the `java.io` package and their subclasses.

5.1.2 Character oriented I/O

Character oriented I/O is used when real people needs to read the data. For this purpose, a `character encoding` is used, in Java this is the `Unicode` character encoding. `Unicode` is an international standard encoding that can represent most of the world's written languages. 16 bits are used to represent one character using this coding schema. In addition, Java can understand UTF, which uses 24 bits for each character.

The classes that are mainly used for this type of I/O are the `Reader` and `Writer` classes in the `java.io` package and their subclasses. Character streams are often wrappers for byte streams to give the end program the desired functionality.

5.2 Overview

The java.io package in Java™2 Platform Standard Edition consists of the interfaces shown in Table A.1, the classes shown in Table A.2 and the exceptions shown in Table A.3 in Appendix A. The classes that are relevant for this thesis, namely the classes that deals with file I/O, are highlighted.

5.3 Streams

An I/O stream represents an input source or an output destination, this can range from disk files or devices to other programs or memory arrays. Such a stream can be anything that can contain data[41]. Different types of data are supported for such streams, mainly bytes, characters and objects. A sequence of one of these data types typically makes a stream.

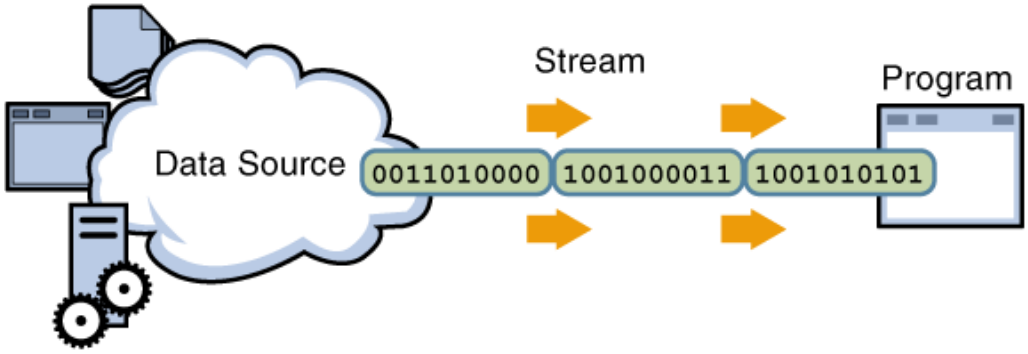

Figure 5.1: Illustration of an input stream

An input stream, as shown in Figure 5.1, reads data from a source that then can be used in an application. An output stream, as shown in Figure 5.2, writes data from an application to a destination.

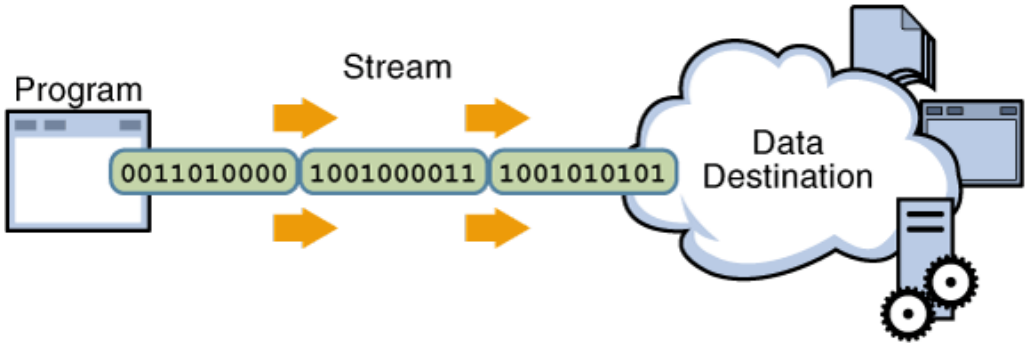

Figure 5.2: Illustration of an output stream

5.3.1 Buffered streams

Writing of small proportions of data can be very expensive in terms of performance. Because of this, buffering should be used to prevent disk reads and writes for each

character[42]. In buffered streams, data is being read from a memory area called the *buffer* instead of being handled directly by the underlying operating system[43].

5.4 Input

The basic method for input in Java I/O is the `read()`-method which is defined in the `InputStream` and `Reader` abstract classes in the `java.io` package and provides reading of a single byte or an array of bytes.

5.4.1 Readers

The different readers in the java.io package perform different read operations as follows ([40], *Section "InputStreams and OutputStreams: InputStreams"*):

- BufferedReader: provides buffering and increases efficiency
  - LineNumberReader: keeps track of how many lines that have been read
- CharArrayReader: reads char arrays
- FilterReader (abstract): modifies the stream as it is read
  - PushbackReader: unreads one or more characters
- InputStreamReader: takes any binary input stream and turns it into a character stream
  - FileReader: connection to an actual file
- PipedReader: used in a Unix pipe
- StringReader: treats a String object as an input device

All these classes are subclasses of the `Reader` class. The sources that can be read from are normally an array of bytes, a file or a pipe. Methods that are used are the `read()` and `close()` methods which reads and closes the stream and frees resources respectively.

File reading

Figure 5.3 shows how file reading is normally done in Java. Catching of exceptions is omitted for the sake of simplicity.

5.5 Output

The basic method for output in Java I/O is the `write()`-method which is defined in the `OutputStream` and `Writer` abstract classes in the `java.io` package and provides writing of a single byte or an array of bytes.

```
FileReader freader = new FileReader(``myfile,txt'');
BufferedReader breader = new BufferedReader(freader);
String s = new String ();
while ((s = breader.readLine()) != null) {
   System.out.println(s);
}
breader.close();
```

Figure 5.3: Example of file reading

5.5.1 Writers

The different writers in the java.io package perform different write operations as follows ([40], *Section "InputStreams and OutputStreams: OutputStreams"*):

- BufferedWriter: provides buffering of output
- CharArrayWriter: provides writing to a character array as if it was a file
- FilterWriter: provides filtering for the data that is going to be written
- OutputStreamWriter:
  - FileWriter: used to write to an actual file
- PipedWriter: the writing end of a Unix pipe
- PrintWriter: used to show internal data in an easily understandable form
- StringWriter: provides writing to a string as if it was a file

All these classes are subclasses of the `Writer` class. The destinations that can be written to are normally an array of bytes, a file or a pipe. Methods that are used are the `write()`, `flush()` and `close()` methods which writes, forces output to be written to destination and closes the stream and frees resources respectively.

File writing

Figure 10.3 shows a typical example of how file writing is done in Java. Catching of exceptions are omitted for the sake of simplicity.

```
File f = new File(``myfile.txt'');
BufferedWriter bwriter = new BufferedWriter(new FileWriter(f));
bwriter.write("A new line");
bwriter.flush();
bwriter.close();
```

Figure 5.4: Example of file writing

5.6 The File class

The File class in the `java.io` package maintains file information (like length of the file and if it is readable among other things). It does not, as the name suggests, refer to an actual file in the file system, but it merely represents either the name of a file or the name of a set of files in a directory. The file might not even exist.

The File class can be used for[44]:

- Creation of directory
- Creation of an entire directory path
- Examination of file characteristics like size, modification date, readable/writable
- Deletion of a file or directory
- Examination whether the File object is a file or a directory
- Renaming of the File object

If an actual file exists at the given location for a File object, the object can be used to perform actions on the file by the use of streams for reading or writing to it.

5.7 New Java I/O

The Java New I/O library, `java.nio`, was introduced in JDK 1.4 with one goal: Increase the speed of I/O operations in the Java language. This is mainly accomplished by introducing operating system structures for I/O, namely channels and buffers. Use of these structures gives an improved speed for mainly file I/O and network I/O. Even if a user does not explicitly include classes from the `java.nio` package, the classes from the `java.io` package have been re-implemented so that they use this new package for optimizing how I/O is done ([45], *Chapter 12: The Java I/O System: New I/O*).

Chapter 6

File System Access Libraries

6.1 The Apache Jakarta Project

The Jakarta Project[46] consists of a multitude of Java open source solutions governed by ASF, the Apache Software Foundation, which is encouraging a collaborative and consensus-based approach for software development.

ASF[47] is an organization that provides organizational, legal and financial contribution for the various Apache open source software projects. The foundation is a membership-based, not-for-profit corporation which tries to ensure the future existence of all of the Apache projects. Its goal is to provide enterprise-grade and freely available software with a large user base. Some of the most known and used software applications from Apache today are the Apache HTTP Server[48] and the Apache Tomcat servlet container[49].

The different Jakarta subprojects are using an open source license for distribution of software. This basically means the following[50]:

- A user is allowed to:
  - Freely download and use Apache software for personal, company internal or commercial purposes
  - use the software from Apache in own projects or in distributions that the user creates
- It is forbidden to:
  - Redistribute the whole or parts of the project without proper attribution
  - Give the impression that the ASF endorses a self-made distribution
  - Give the impression that the user himself created the software
- In addition, the license requires the user to:
  - Include a license copy in any redistribution that includes Apache software
  - Provide clear attribution to ASF in case of use

This thesis has examined and used two Jakarta subprojects, namely the Slide project and the Commons VFS project.

6.2 Slide WebDAV library

The Slide WebDAV[51] library is a content repository that can act as a basis for a content management system. This description of Slide will concentrate on the web-based content management that can be achieved by using the WebDAV client library[52] that is included in Slide.

6.2.1 The WebdavResource class

The main class used in the Slide WebDAV client library is the `WebdavResource`[53] class. Unfortunately, the documentation for this client library is quite poor, so in order to find out how it works, a test implementation had to be done, which will be further described in Chapter 7.3.

Constructors

Several constructors for this class exist; however, the one that has been used throughout this thesis is the one shown in Table 6.1.

Constructor definition	Explanation
`WebdavResource(HttpURL httpURL)`	Creates a connection to the WebdavResource specified by the httpURL input parameter

Table 6.1: Constructor used from the WebdavResource class

The `HttpURL` class is part of the `org.apache.commons.httpclient`[54] package from the Jakarta Commons project. `HttpClient` is a framework for working with the client side of the HTTP protocol. For security reasons, throughout this thesis the subclass of the `HttpURL`, namely `HttpsURL`, is used for input to the WebdavResource constructor.

Methods

Table 6.2 shows the most important methods in the WebdavResource class, their return type and their explanation. Many of the methods correspond to the HTTP and WebDAV methods described in Chapters 3.1.4 and 4.5. For many of these methods, different method definitions with different input parameters that enables other more specific usages are also available.

6.2.2 Patching of the library

During work with the test implementation of the Slide WebDAV library, it was discovered a bug which made listing of children elements of a directory WebdavRe-

Returns	Method definition	Explanation
boolean	`copyMethod()`	Execution of the COPY method
boolean	`deleteMethod()`	Excecution of the DELETE method for this resource
boolean	`exists()`	Checks whether this WebdavResource exists or not
boolean	`exists()`	Checks whether this WebdavResource exists or not
boolean	`getMethod()`	Execution of the GET method which populates the input file with the data from the WebdavResource
String	`getPath()`	Gets the path of the resource
boolean	`headMethod()`	Execution of the HEAD method
boolean	`isCollection()`	Checks whether the resource is a directory or not
WebdavResource[]	`listWebdavResources()`	Returns an array with the children resources of this resource
boolean	`lockMethod()`	Execution of the LOCK method
boolean	`mkcolMethod()`	Execution of the MKCOL method
boolean	`moveMethod()`	Excecution of the MOVE method
boolean	`optionsMethod()`	Execution of the OPTIONS method
Enumeration	`propfindMethod()`	Execution of the PROPFIND method
boolean	`proppatchMethod()`	Execution of the PROPPATCH method
boolean	`putMethod()`	Execution of the PUT method
boolean	`unlockMethod()`	Execution of the UNLOCK method

Table 6.2: Important methods from the WebdavResource class

source incomplete. To correct this, patching of one of the methods had to be done. This procedure is described in Appendix B.

6.3 The VFS library

The Virtual File System[55] from Jakarta Commons provides a way to access different file systems through a single API. When used, it provides a uniform view of files from different sources, for example from a local disk, an HTTP server or a WebDAV compliant server.

VFS has built-in support for various file system types, like[56]:

- FTP

  - Gives access to local files on the local file system
  - `[file://] absolutepath`
  - **Example:** `file:///home/someuser/somedir`

- zip, jar and tar
  - Gives read-only access to the zip, jar and tar compressed file types
  - `zip://archive-file-URI[! absolute-path]`
  - `jar://archive-file-URI[! absolute-path]`
  - `tar://archive-file-URI[! absolute-path]`
  - **Example:** `jar:../lib/classes.jar!/META-INF/manifest.mf`

- gzip and bzip2
  - Gives read-only access to the contents of gzip and bzip2 files
  - `gz:// uri-to-compressed-file`
  - `bz2:// uri-to-compressed-file`
  - **Example:** `gz://my/gz/file.gz`

- HTTP and HTTPS
  - Gives access to the files on a HTTP server
  - `http://[ username[: password]@] hostname[: port][ absolute-path]`
  - `https://[ username[: password]@] hostname[: port][ absolute-path]`
  - **Example:** `https://myusername:mypassword@somehost:8080/index.html`

- WebDAV
  - Gives access to files on a WebDAV server
  - `webdav://[ username[: password]@] hostname[: port][ absolute-path]`
  - **Example:** `webdav://somehost:8080/test`

- FTP
  - Gives access to files on a FTP server
  - `ftp://[ username[: password]@] hostname[: port][ absolute-path]`
  - **Example:** `ftp://myusername:mypassword@somehost/downloads/somefile.txt`

- SFTP
  - Gives access to files on a SFTP server
  - `sftp:// [ username[: password]@] hostname[: port][ absolute-path]`
  - **Example:** `sftp://myusername:mypassword@somehost/downloads/somefile.txt`

- CIFS
  - Gives access to files on a CIFS server, such as a Samba server
  - `smb://[ username[: password]@] hostname[: port][ absolute-path]`
  - **Example:** `smb://somehost/home`

- Temporary Files
  - Gives access to a temporary file system that is deleted when VFS shuts down
  - `tmp://[ absolute-path]`
  - **Example:** `tmp://dir/somefile.txt`

Chapter 7

Library Test Implementations

7.1 General

The purpose of developing test implementations of possible libraries was to learn more about how these libraries worked in the environment at CERN. Slide WebDAV Library was a natural library to start with, since it is the one Java-based library that is listed on the WebDAV page[57]. The possible use of VFS was discovered later in the process by Artur Wiecek, one of my supervisors at CERN, and examined at a later stage of the permanent storage project, after the test implementation of Slide was finished and proved itself as a working possibility.

7.2 Test environment

It was decided early in the thesis project that the Distributed File System (DFS), the main file system at CERN, was the most usable place to let users store and access their files. This was done for several reasons:

- DFS is easily accessible from both CERN and locations outside of CERN by the use of the web interface at `https://dfs.cern.ch` (internal web page not accessible to people without correct credentials)
- Most people affiliated with CERN has an account with a personal area to store files at DFS
- A WebDAV interface is provided to it, so connecting with a WebDAV library should work

The test implementations themselves were done from a local computer equipped with Java and a Tomcat container similar to the ones used at the J2EE Public Service inside of CERN's private network. Since the HTTP and HTTPS protocols were used, location was not an issue because such traffic is cleared through the CERN firewall.

7.3 Slide

The test implementation of Slide was done as a "Proof-of-concept"[58] to show that the Slide WebDAV library could be used in such an environment described above.

Mainly, the test implementation works as a simple command line file client, with the following commands available:

- `list`: lists all the files and directories in the working directory
- `createfile`: creates a file from user input in the working directory
- `todir`: makes another directory the working directory, whole directory path must be specified
- `quit`: quits the command line interface
- `createdir`: creates a new directory in the working directory
- `save`: saves the specified file from the DFS working directory to the local file system on the local computer running the command line interface
- `import`: imports a file from the local file system of the local computer and stores it at the working directory at DFS
- `show`: shows the content of the specified file on DFS

7.3.1 How does the implementation work

The following subsections gives example code on how the above commands are implemented using the constructor and methods described in Chapter 6.2.1.

Connecting to the DFS location

Figure 7.1 shows how connection to a given location at the DFS file system at CERN is done through the use of the classes `org.apache.commons.httpclient.HttpsURL` and `org.apache.webdav.lib.WebdavResource`. Most of the code examples below needs `try` and `catch` blocks to catch `HttpException`s and `IOException`s to work correctly, but for the sake of simplicity these blocks are omitted. For the examples of the different commands below, the WebdavResource created in Figure 7.1 is used as the `wdr` reference.

```
HttpsURL hrl = new HttpsURL("https://dfs.cern.ch/dfs/Users/h/holmesta/");
hrl.setUserinfo("username", "password");
wdr = new WebdavResource(hrl);
```

Figure 7.1: Connection to DFS using the HttpsURL and WebdavResource classes

The `list` command

Figure 7.2 shows the methods used by the `list` command, the `listWebdavResources()` and the `isCollection()`-methods.

```
ArrayList<WebdavResource> directories = new ArrayList<WebdavResource>();
ArrayList<WebdavResource> files = new ArrayList<WebdavResource>();

WebdavResource[] resources = wdr.listWebdavResources();

//loops through all the resources found by the listWebdavResources()
//call and places them in the appriopriate ArrayList

for (int i = 0; i < resources.length; i++) {

   if (resources[i].isCollection())

      directories.add(resources[i]);

   else

      files.add(resources[i]);

}
```

Figure 7.2: Example code for listing of files and directories

The `createfile` command

Figure 7.3 shows the basic methods used by the `createfile` command, the `putMethod()` and the `getPath()`-methods.

```
//the fileName and contents string variables are
//already initialized as input from the user

if (wdr.putMethod(wdr.getPath()+"/"+fileName, contents))

   System.out.println("File "+fileName+" successfully created");

else

   System.out.println("No file created");
```

Figure 7.3: Example code for creating a file

The `todir` command

The `todir` command performs the same action as shown in Figure 7.1, but gives as input to the `HttpsURL`-constructor the path to the desired directory.

The `createdir` command

Figure 7.4 shows the basic methods used by the `createdir` command, the `mkcolMethod()` and the `getPath()`-methods.

```
//the dirName string variable is already
//initialized as input from the user

if (wdr.mkcolMethod(wdr.getPath()+dirName))

   System.out.println("Directory "+wdr.getPath()+dirName+" created");

else

   System.out.println("No directory created");
```

Figure 7.4: Example code for creating a directory

The `save` command

Figure 7.5 shows the basic method used by the `save` command, the `getMethod()`-method.

```
//the remotePath and localPath string variables
//are already initialized as input from the user

if (wdr.getMethod(remotePath, new java.io.File(localPath)));

   System.out.println("File successfully saved at location "+localPath);

else

   System.out.println("File not saved. Check that both paths are valid");
```

Figure 7.5: Example code for saving a file to the local file system

The `import` command

Figure 7.6 shows the basic methods used by the `import` command, the `putMethod()` and `getPath()`-methods.

```
//the localPath string variable is already
//initialized as input from the user

File f = new File(localPath);

if (wdr.putMethod(wdr.getPath(), f))

   System.out.println("Successfully import to DFS");

else

   System.out.println("Import did not work");
```

Figure 7.6: Example code for importing a file to the DFS location

The `show` command

The `show` command displays the working directory of the `WebdavResource` by use of the `getPath()`-method.

7.4 VFS

A simple command line client, which with small modifications made it possible to test whether VFS could be used, is provided as an example[59] together with the VFS source code. As with the self-produced command line client for the Slide test implementation, the following operations are provided in the VFS command line client:

- `cd [folder]`: changes working directory
- `cp <src> <dest>`: copies a file or a folder
- `cat <file>`: displays the content of a file
- `help`: shows a help message
- `ls [-R] [path]`: lists contents of a file or a folder
- `pwd`: displays the working directory
- `rm <path>`: Removes a file or a folder
- `touch <path>`: Sets the `last-modified` value for a file

7.4.1 How

To get access to files and create file systems in VFS, the `FileSystemManager`[60] class is used. Using this class, a `FileObject` can be created using the `resolveFile()`-method with an input parameter formatted as described in Chapter 6.3. Figure 7.7 shows the use of these classes in the command line interface trying to access a specific location at DFS.

```
FileSystemManager mgr = VFS.getManager();
FileObject cwd = mgr.resolveFile("https://holmesta:password"
                 +"@dfs.cern.ch/dfs/Users/h/holmesta/");
```

Figure 7.7: Example code for accessing DFS through VFS

The following methods from the `FileObject`[61] class are used to perform the commands described above:

- `delete(FileSelector selector)`: Deletes internal members of the `FileObject` that matches the input selector
- `copyFrom(FileObject srcFile, FileSelector selector)`: Copies another file and its internal members that match the input selector to this `FileObject`
- `getChildren()`: Lists the children of this `FileObject`
- `getContent()`: Returns the content of the `FileObject`
- `exists()`: Checks if the file exists

7.4.2 Tests of different connections

To test the flexibility of the VFS library, different connections to DFS was tested by changing the input parameter of the `resolveFile()`-method.

HTTP connection to DFS

The input `http://holmesta:mypassword@dfs.cern.ch/dfs/Users/h/holmesta/` to the `resolveFile()`-method tries to make a normal connection to the author's public area at DFS. This connection works perfectly, there are no problems with performing any of the commands listed above. However, this connection is not a **secure** connection, because the password is transmitted in cleartext. This makes such a connection practically unusable.

WebDAV connection to DFS

The input `webdav://holmesta:mypassword@dfs.cern.ch/dfs/Users/h/holmesta/` was used as input to `resolveFile()` to test the WebDAV connectivity of VFS. This is using the Slide WebDAV library, so in theory it should work. The connection performed the commands given in the command line interface as expected, but as for the HTTP connection this is not a **secure** connection considering that the password is sent in cleartext.

Secure HTTPS connection to DFS

The input `https://holmesta:mypassword@dfs.cern.ch/dfs/Users/h/holmesta/` to the `resolveFile()`-method was not successful. The VFS was not able to connect to this DFS location in a secure manner, and further investigation yielded no success or easy solution to this problem.

CIFS connection to DFS

`smb://holmesta:mypassword@cerndfs01.cern.ch/dfs/users/h/holmesta/` was also tried as input parameter to the `resolveFile()`-method, but this was not successful and it was not possible to achieve a connection. Some time was spent trying to understand what was causing the malfunctioning, but no apparent solution was found.

Chapter 8

Research Conclusions

8.1 Slide evaluation

Slide, through the WebDAV client library, has been found to have the following advantages:

- Working support for all wanted file operations, like getting a file, putting a file, getting information about properties of files and directories and creation/removal of directories
- Through the use of the `HttpsURL` class it provides a secure connection where passwords are protected

On the other hand, some weak points have also been detected:

- The documentation is poor and it has required a lot of time and effort to understand how to use the library
- It seems that further progress has stopped, as the newest version of the library has not managed to correct the bug described in Appendix B
- Performance problems which has not been a topic for this thesis

8.2 VFS evalution

VFS has been found to have the following advantages:

- As for Slide, working support for all file operations
- It looks like development of VFS is more active than development of Slide
- A lot of other file systems are supported, as described in Chapter 6.3, which gives more flexibility concerning future development of permanent storage for the J2EE Public Service

- Better documentation than Slide

Subsequently, some quite important findings have been done when it comes to weak points:

- In the CERN environment using DFS as the main storage medium, no secure access is possible to obtain
- CIFS connection to DFS was not possible and the author did not manage to find out why this was not working

8.3 Overall evaluation

The most ideal solution for this thesis work would be to find a way to directly manipulate the Java source code, such that web application authors hosting their applications at the J2EE Public Service could continue to use the normal packages for Java I/O. This is not possible since the source code for the Java API is not available, so different solutions had to be investigated.

Creating a library that the web application author can include in the web application file deployed to the J2EE Public Server was chosen as approach. This basically means that the idea mentioned in the thesis description about capturing file requests and redirect them to another, possibly remote, location is used. This remote location is then the DFS file system and redirection happens in the library especially created for the DFS connection.

A Master thesis has a natural limit on the amount of time available for research and investigation of unexpected results (like the malfunctioning of the VFS library for DFS). It was therefore decided, in collaboration with supervisors and more experienced people at CERN, that the Slide solution was the correct choice for implementing a permanent storage solution for the J2EE Public Service.

Despite the fact that documentation is poor and the impression that the project itself is no longer well maintained, it was strongly believed that a working solution based on Slide which was secure so that user's passwords were not compromised in any way, was the correct path to follow for further development.

VFS was mainly rejected because of the lack of security when connecting through HTTP and WebDAV. If this had worked, VFS would clearly have been a better choice considering the wider choice of options it gives for future development of permanent storage for the J2EE Public Service.

Part III

Design, Implementation & Testing

Chapter 9

Implementation of stand-alone class File.java

9.1 Background

The File.java class was meant to provide an easy way for users of the J2EE Public Service to access their files at DFS. The class extends the `java.io.File` class, hereby inheriting all the functionality provided by it.

9.1.1 Functionality of File.java

The basic methods that can be used to achieve file storage and retrieval from this class are:

- `save()`
- `retrieve()`
- `save(String location, String username, String password)`
- `retrieve(String location, String username, String password)`

For the two last methods to succeed, a valid location to DFS has to be given as input parameter along with valid username/password for this location. The two first methods depends on the existance of a `properties` file in the `WEB-INF/classes/` directory of the web author's web application, further described in Chapter 9.1.2.

9.1.2 User WAR-file construction

Chapter 2.2 described how the J2EE Public Service works concerning how users deploy their created WAR files. To make use of the File.java class described in this chapter in such a web application, the user needs to add a .jar file especially created for this purpose. This .jar file (included in the electronical documentation of this thesis with the name `Standalone.jar`) bundles all the necessary libraries that the File.java class needs to work properly including the class itself into one package:

- The Jakarta Slide WebDAV client library as described in Chapter 6.2
- The JDOM[62] library for xml parsing
- The Jakarta Commons HttpClient[54] package for HTTPS connections
- The `j2ee.permanent.storage.File` class itself

In addition, when using the `save()` and `retrieve()` methods, the user has to provide a `properties` file in the WAR-file which is called `dfs.properties`. This file has to include three keys in it, which is used by the `save()` and `retrieve()` methods to create the DFS connection.

This `dfs.properties` file should include the following key-value pairs:

- `dfs.location`: Specifies the path to the users home area at DFS
- `dfs.username`: The username of the user
- `dfs.password`: The password of the user

9.2 Requirements

The requirements for this class were the following:

- 1: Make it possible for users of the J2EE Public Service to retrieve and/or save files from DFS through their web application
- 2: Make saving/retrieval secure
- 3: Make saving/retrieval as transparent as possible for the users
- 4: Make sure that a web application only can access files from the web application author's area at DFS

9.3 System design

The File.java stand-alone class is a very simple library that, by extending `java.io.File`, provides a simple way of saving and retrieval of files from DFS. It is using the classes `org.apache.commons.httpclient.HttpsURL` and `org.apache.webdav.lib.WebdavResource` to achieve this.

Figure 9.1 shows a sequence diagram for a successful call to the `retrieve()`-method when a valid `File` object already has been created. First, the call to the private method `getLocationInfo()` will store the key-value pairs from the `dfs.properties` file as member variables of the class and use them to create an `HttpsURL` and as input parameters for the `setUserInfo()`-method. A WebdavResource is created on the basis of the `HttpsURL` object, and if a file exists at DFS with the same pathname as the `File` object was created with, the `getMethod()` described in Chapter 6.2 will

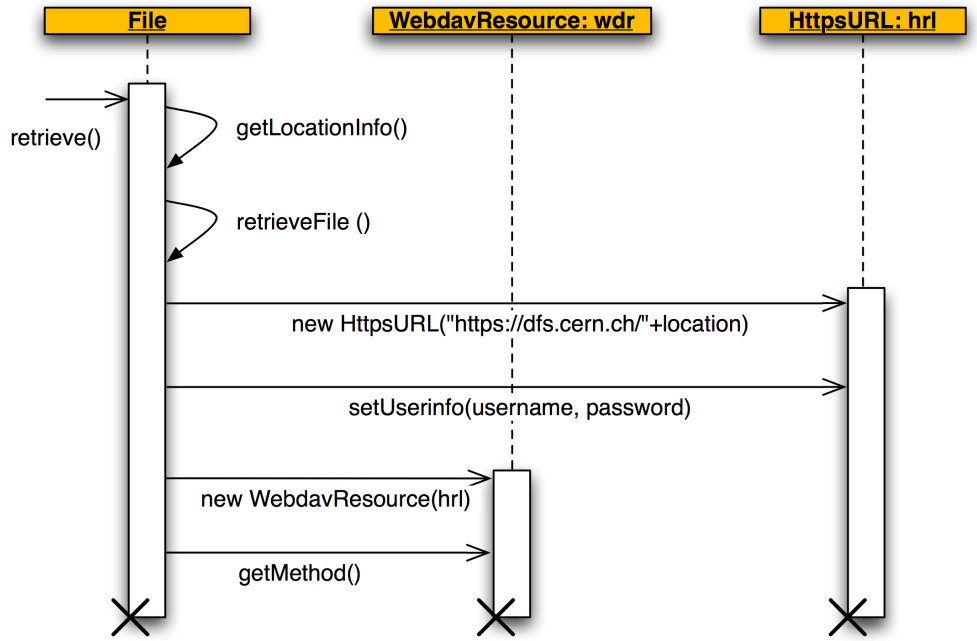

Figure 9.1: Sequence diagram for a successful `retrieve` operation

transfer the file to the `temporary` directory of the user container at the J2EE Public Service and the file can be further manipulated.

The same file can then be written back to DFS using the `save()`-method which is using the `putMethod()` described in Chapter 6.2. A part from the change of methods, it can be seen from Figure 9.2 that the same operations as described above are used for the save operation as for the retrieve operation.

9.4 Evaluation

Due to the simple nature of the File.java class, not much testing needed to be done to make sure that the functionality of the class is as expected. However, some advantages and limitations were discovered during the process, these are described below.

9.4.1 Advantages

Simple and easy to use

The File.java class provides simple and easy-to-use functionality for retrieval and saving of files to or from DFS. Mainly two methods are available to the web application developers using the J2EE Public Service, namely the `save()` and `retrieve()` methods, and their behavior is easily understood from the documentation provided with the class.

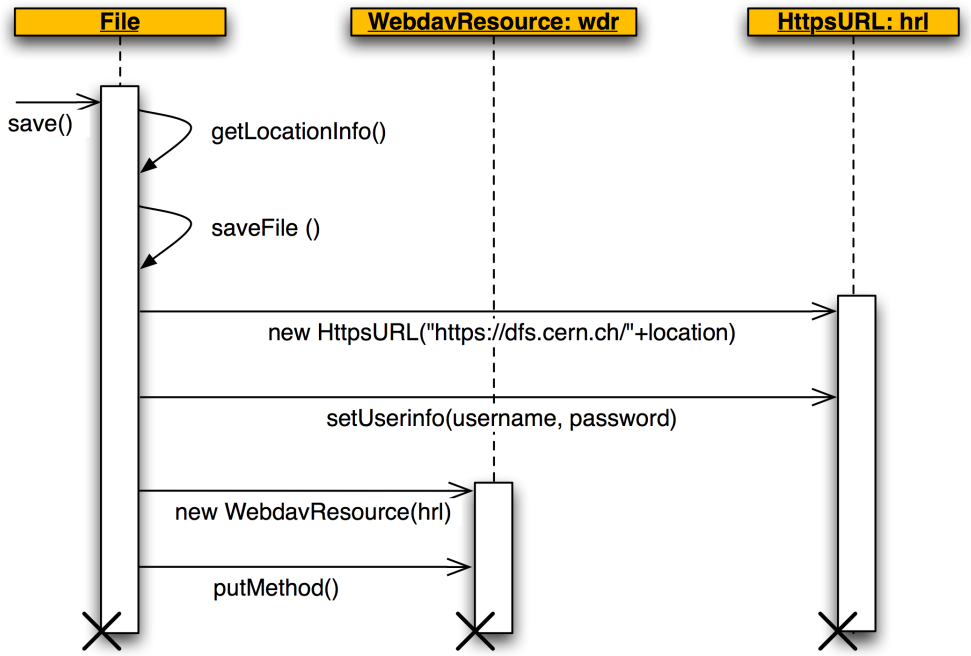

Figure 9.2: Sequence diagram for a successful `save` operation

Extending the java.io.File class

By extending the `java.io.File` class, all methods from this class are directly available to use with the `File` objects created.

Security

By the use of the `HttpsURL` class when creating a `WebdavResource` connection to DFS, the password of different users is protected against eavesdropping.

File protection

The overall concept of the J2EE Public Service with each container only having write/read access to its own `temporary` directory makes it impossible for another web application running on another container at the J2EE Public Service to access files. Security issues were described in Chapter 2.2 and 2.3.

9.4.2 Limitations

Transparency

To use the retrieval/store possibilities of the File.java class, users have to perform other method calls than usually done in the Java language, as described in Chapter 5.4 and 5.5. This means that the users have to learn new ways of dealing with Java

I/O that might not be compatible with old applications in one way or another, and this breaks with requirement 4 described in Chapter 9.2

No file hierarchy allowed

For the File.java class, only one of the constructors of the `java.io.File` class was implemented, namely the most simple one which only takes a pathname to the requested file as an input.

Currently, the File.java class does not support pathnames with parents, which means that any File object creation that involves a parent directory (such as `new File("mydirectory/myfile.txt")` will fail when the `retrieve()` or `save()` methods are called. This is because these two methods do not create any non-existing parent directories neither at the local temporary directory nor at DFS.

9.4.3 Overall evalution

For simple use, the File.java class is sufficient and it satisfies most of the criterias in Chapter 9.2. However, because of the lack of support for file hierarchy and the lack of transparency for the users, a new and more extensive solution was needed. This solution is outlined in the next chapter, where a re-implementation of the `java.io` package is described.

Chapter 10

Re-implementation of the java.io package

From the evaluation done in Chapter 9.4, there was a need to create a library implementation that addressed the weaknesses found. The goal was to do an extensive re-implementation of the classes that deals with File I/O in the `java.io` package.

This chapter describes the work that has been done with this new package, called `j2ee.io`, and shows how it works through sequence diagrams for read and write operations, an overview of the different methods implemented for the `j2ee.io.File` class and a code example from the `j2ee.io.FileWriter` class.

10.1 General

The `j2ee.io` package contains extended versions of almost all the classes in the `java.io` package, except classes that are implemented as `final` and because of this can not be extended or are deprecated and because of this should not be extended.

An overview of the classes where no extended versions have been created is given in Table 10.1.

Class name	Reason
FileDescriptor	final
FilePermission	final
LineNumberInputStream	deprecated
ObjectStreamClass	not public
SerializablePermission	deprecated
StringBufferInputStream	deprecated

Table 10.1: Classes from `java.io` not implemented in `j2ee.io`

Some classes have been given special treatment because of integration with file retrieval/storage from DFS. These classes are the `j2ee.io.File` class, the `j2ee.io.FileWriter` class and the `j2ee.io.FileOutputStream` class and they will be closer described in Chapter 10.1.1 and 10.1.3.

All the other classes from the `java.io` package have only been included in the `j2ee.io` package because of usage considerations for the J2EE Public Service web application authors. All they do is to call the constructors of their superclass with exactly the same arguments, merely a copy of their superclasses. This is done to allow the web application authors to import only one whole package during code writing, the `j2ee.io` package, instead of some classes from the `java.io` package and some classes from the `j2ee.io` package.

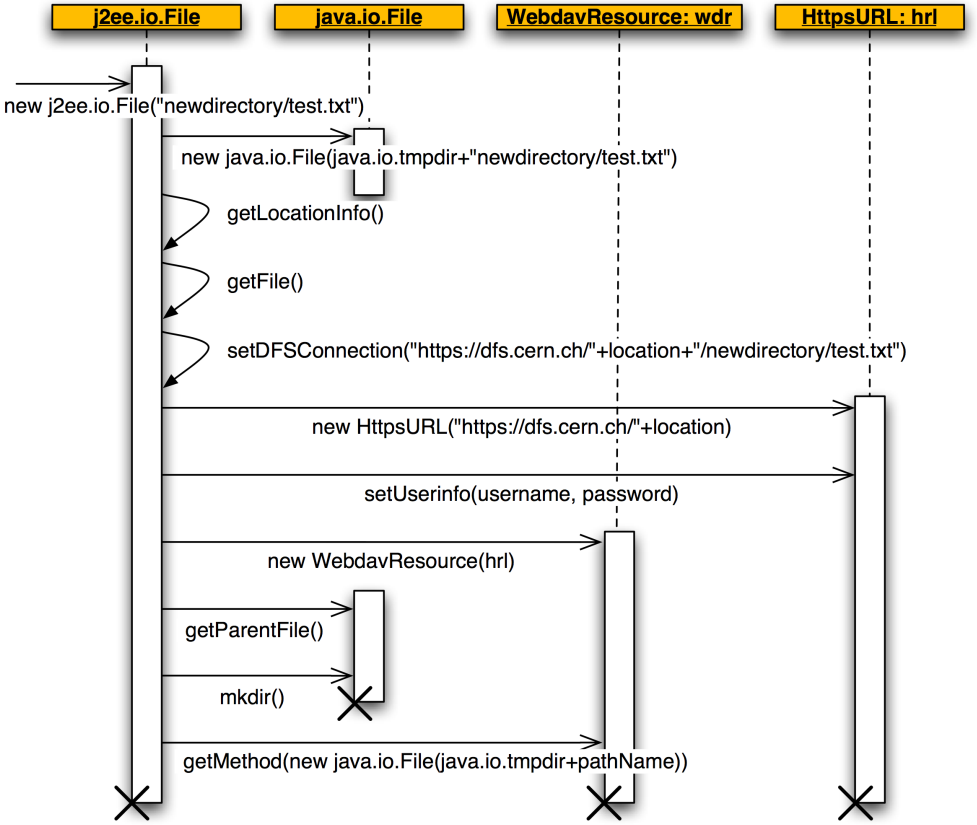

Figure 10.1: Sequence diagram for creation of a `j2ee.io.File` object existing at DFS

10.1.1 The j2ee.io.File class

The `j2ee.io.File` class extends the `java.io.File class` described in Chapter 5.6. It overloads most of the constructors and methods found in its superclass.

If the web application author has included a `dfs.properties` file in the `WEB-INF/classes/` directory of his web application as described in Chapter 9.1.2, the `j2ee.io.File` class will try to connect to DFS through the use of the `WebdavResource` class. If the input to the specific constructor specifies a **file**, then the class will use the `getMethod()` to retrieve the file from DFS and place it in the temporary directory at the container hosting the web application, including creating any non-exisiting parent directories on the way. Figure 10.1 shows a sequence diagram describing this functionality.

If no properties file is included in the web application deployed by the user, the constructors and all the methods in the `j2ee.io.File` class will work as normally expected in the `java.io.File` class.

Constructor definition	Explanation
`File(String pathName)`	Creates the File object in the temporary directory from the `pathName` variable
`File(String parent, String child)`	Creates the File object in the temporary directory from the input `parent` and `child` variables
`File(File parent, String child)`	Creates the File object in the temporary directory from the input `parent` and `child` variables

Table 10.2: Constructors implemented for the `j2ee.io.File` class

Constructors

Table 10.2 shows the constructors available for the `j2ee.io.File` class.

Methods

Table 10.3 shows the public methods available for the `j2ee.io.File` class and their explanations. The explanations given assume that a properties file exists, if not, the methods will behave as explained in the Java API for the `java.io.File`[63] class.

10.1.2 File reading

File reading in java, described more closely in Chapter 5.4.1, does not need any special treatment to work as expected when using the `j2ee.io` package. If a given file exists at DFS and the `dfs.properties` file exists with correct credentials in the web application of a user, whenever a new `j2ee.io.File` object is created referring to this file, it will be copied to the temporary directory of the web application's container and be readable by normal Java file reading methods.

10.1.3 File writing

File writing in Java is done as described in Chapter 5.5.1. In the `j2ee.io` package, the classes `FileWriter` and `FileOutputStream` has been specially designed to deal with file writing to DFS when a `dfs.properties` file is available. This is done by overloading the `flush()`-method of the two classes, and this method is implemented identically in both places.

Figure 10.2 shows a typical file write example using the `j2ee.io.FileWriter` class and a call to the `flush()`-method to write the changes (In Figure 10.2 this adds a line with "testing" to the file) done to disk. This example shows creation of a

Returns	Method definition	Explanation
boolean	`delete()`	Deletes the file or directory at DFS
boolean	`exists()`	Checks if the file or directory exists at DFS
java.io.File	`getAbsoluteFile()`	Returns a new File object from the absolute path of this File object
String	`getAbsolutePath()`	Returns the absolute pathname string from DFS
java.io.File	`getCanonicalFile()`	Returns a new File object from the canonical path of this File object
String	`getCanonicalPath()`	Returns the canonical pathname string from DFS
String	`getName()`	Returns the name of the file or directory
String	`getParent()`	Returns the pathname string at DFS for this File object's parent
java.io.File	`getParentFile()`	Returns the File object for this File object's parent
String	`getPath()`	Returns the pathname of this File object
boolean	`isAbsolute()`	Tests if this abstract pathname is absolute
boolean	`isDirectory()`	Tests if this File object is a directory
boolean	`isFile()`	Tests if this File object is a file
String[]	`list()`	Lists the children of this File object if it is a directory
String[]	`list(FilenameFilter filter)`	Lists the children that satisfies the `filter` of this File object
File[]	`listFiles()`	Returns an array with File objects created from the pathnames returned by the `list()` method
File[]	`listFiles(FilenameFilter filter)`	Returns an array with File objects created from the pathnames returned by the `list(FilenameFilter filter)` method
boolean	`mkdir()`	Creates the directory at DFS if the File object is a directory
boolean	`mkdirs()`	Creates the directory at DFS including subdirectories if the File object is a directory

Table 10.3: Methods from the `j2ee.io.File` class

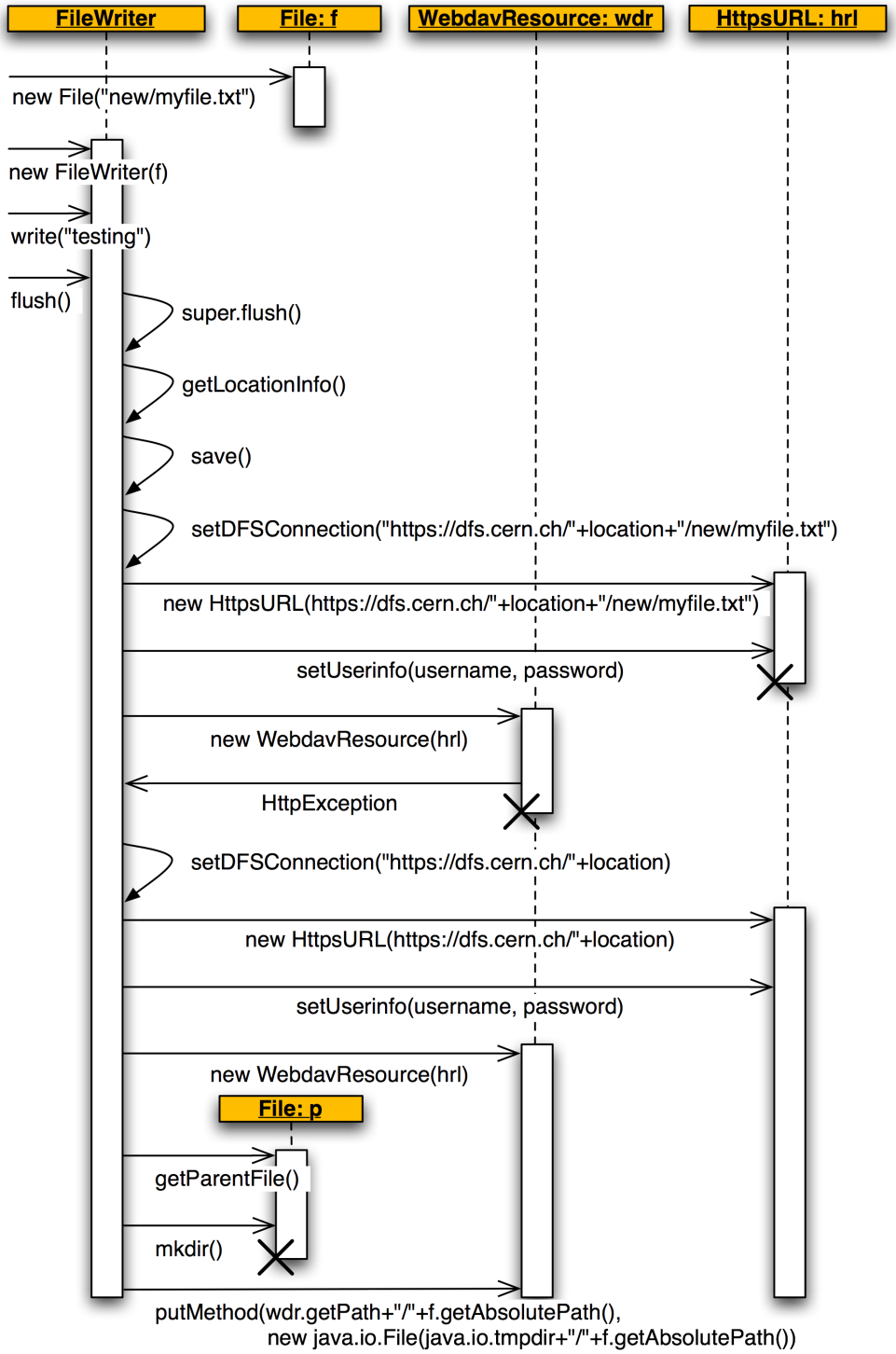

Figure 10.2: Sequence diagram for file writing using the `j2ee.io` package

new file called **myfile.txt** in the directory **new** which does not exist at DFS. A code example showing the `save()` method (which is called in the beginning of the `flush()` method in Figure 10.2) is shown in Figure 10.3, with explanation below.

Code explanation

In Figure 10.3 the `save()` method, which is the method that provides the store functionality for the `j2ee.io.FileWriter` and `j2ee.io.FileOutputStream` classes, is shown.

The code example is corresponding to the code run after the call of the `save()` method in the sequence diagram in Figure 10.2. Firstly, the `setDFSConnection()` on line **3** will prepare for a WebdavResource creation on the **new/myfile.txt** location of the DFS base directory. The creation of a new `WebdavResource` on line **6** will then throw an `HttpException` because the file does not exist at DFS, which will make the program jump to line **22** to catch this exception.

The `setDFSConnection()` method is then called again on line **25**, but this time it only prepares for a `WebdavResource` creation on line **39** to the DFS base directory. Before this new creation, the parent of the `File` object is retrieved on line **27**, which in this case will be a `j2ee.io.File` object referring to the **new** directory. Since this object is not null and all parent directories of the file to be written already exists locally (otherwise, the `java.io.FileWriter` constructor throws a fatal `FileNotFoundException`), the call to the `mkdir()` method on line **38** is to assure that the directory exist at DFS before the `putMethod()` is called on line **40** with the path to the new file as first argument and a new `java.io.File` object referring to the already locally flushed file as the second argument.

10.2 Requirements

The requirements of the `j2ee.io` implementation are the same as described in Chapter 9.2. However, there is an increased focus on transparency for the web application authors and on providing similar functionality for file I/O as the `java.io` package.

In addition, addressing the issue that the stand-alone File.java class created concerning storing and retrieving files from subdirectories has been a new requirement.

10.3 Testing

The `j2ee.io.File` java class has been tested for the following scenarios:

- Creation of a file object in the root directory
- Creation of a file object in subdirectories (and generally for any subdirectory of any other subdirectory) that does not exist at DFS
- Creation of file object in subdirectories (and generally for any subdirectory of any other subdirectory) that exist at DFS

```
1  private void save() {
2
3   if (setDFSConnection("https://dfs.cern.ch/"+location+"/"
4                        +file.getAbsolutePath())) {
5    try {
6     WebdavResource wdr = new WebdavResource(hrl);
7
8     //if we get here, it means that file does exist at DFS
9     try {
10     setDFSConnection("https://dfs.cern.ch/"+location);
11     wdr = new WebdavResource(hrl);
12     wdr.putMethod(wdr.getPath()+"/"+file.getAbsolutePath(), new java.io.File(
13             System.getProperty("java.io.tmpdir")+"/"+file.getAbsolutePath()));
14    }
15    catch (HttpException httpe) {
16      return;
17    }
18    catch (IOException ioe) {
19     return;
20    }
21   }
22   catch (HttpException httpe) {
23
24    //if we are here, this means that the file does not exist
25    if (setDFSConnection("https://dfs.cern.ch/"+location)) {
26     try {
27      java.io.File parent = file.getParentFile();
28
29      //if no parent we don't have to make any directories locally
30      if (parent == null) {
31       WebdavResource wdr = new WebdavResource(hrl);
32       wdr.putMethod(wdr.getPath()+"/"+file.getAbsolutePath(), new java.io.
33         File(System.getProperty("java.io.tmpdir")+"/"+file.getAbsolutePath()));
34       return;
35      }
36
37      //if one parent, we should create this one directory locally
38      else if (parent.mkdir()) {
39       WebdavResource wdr = new WebdavResource(hrl);
40       wdr.putMethod(wdr.getPath()+"/"+file.getAbsolutePath(), new java.io.
41         File(System.getProperty("java.io.tmpdir")+"/"+file.getAbsolutePath()));
42       return;
43      }
44
45      //if several parent directories, we have to create them all
46      else if (parent.mkdirs()) {
47       WebdavResource wdr = new WebdavResource(hrl);
48       wdr.putMethod(wdr.getPath()+"/"+file.getAbsolutePath(), new java.io.
49         File(System.getProperty("java.io.tmpdir")+"/"+file.getAbsolutePath()));
50       return;
51      }
52
53      else
54       return;
55     }
56     catch (HttpException httpe2) {
57      return;
58     }
59     catch (IOException ioe) {
60      return;
61     }
62    }
63   }
64   catch (IOException ioe) {}
65   return;
66  }
67 }
```

Figure 10.3: Code sample from the `save()` method

- Creation of file object when there is no `dfs.properties` file to provide credentials for DFS (Behavior is the same as for the `java.io.File` class)
- Method functionality for all methods for all above cases

The `j2ee.io.FileWriter` and `j2ee.io.FileOutputStream` has been tested with `j2ee.io.File` objects created from the four upper cases above and seem to store the files for all these cases when the `flush()` method is called.

It has to be noted that testing is only done in one specific environment, namely the author's environment at CERN. It has not yet reached production for the J2EE Public Service, but has been provided for users who have specifically asked for it so that they can test it out and report any problems found.

10.4 Evaluation

10.4.1 Advantages

Provides a whole package that can be used for file retrieval/storage

By extending the `java.io` package, this implementation provides similar functionality with the exception that file retrieval and storage is done automatically if the web application author has included a `dfs.properties` file in the WAR-file containing the web application.

Transparency for the user

The web application author can use the same methods as in normal Java I/O and does not have to adapt any code written on an earlier stage to a new way of doing I/O for the `j2ee.io` package to work.

File hierarchy allowed

A web author can access and write to any file in any directory below the directory specified with the key `dfs.location` in the `dfs.properties` file.

Security & File protection

The same considerations as in Chapters 9.4.1 and 9.4.1 is valid for the `j2ee.io` package implementation as well.

10.4.2 Limitations

File locking and the lost update problem

File locking is a feature of the Slide WebDAV client library, but has not been used during the development of the `j2ee.io` package because of the stateless nature of file I/O in Java. There is no way of enforcing unlocking of resources if a resource has been locked, because it is not possible to know when a web application is finished editing a resource. This could have been done by providing specific methods for file locking and unlocking, but this would leave unlocking of locked resources to the web application authors which would not be a good solution.

Not enough testing

As described in Chapter 10.3, there has been some limitations when it comes to testing the `j2ee.io` library in a user environment, mostly due to time constraints. It is not possible to guarantee operation in a production environment before this is done.

10.4.3 Overall evaluation

Overall, the `j2ee.io` package provides a solution that enables file reading in the J2EE Public Service from all different locations in a web author's DFS hierarchy in a transparent and secure way. For file writing, some more care has to be taken because there is no file locking implemented. Lost updates can happen if the web application author uncritically opens up for file writing for all users of the web application in question.

Conclusions

The development of a permanent storage solution for the J2EE Public Service has been done through a research phase where possible ways of using WebDAV to achieve file retrieval and storage was investigated, followed by an implementation phase where two different solutions are provided for the users of the J2EE Public Service.

During the research phase, two different libraries providing access to file systems were investigated and test implementations were made to see if these libraries worked in the specific CERN environment. The Slide WebDAV library provided a working and secure connection through WebDAV to the DFS file system. An investigation of VFS, a library that was supposed to be able to connect to several types of file systems and thereby offering enhanced functionality, however found that VFS was problematic to use as connection to DFS.

The stand-alone File.java class developed during the work's first phase was after evaluation found to be too simple and did not provide the key functional elements described in the problem formulation. Further work and thoughts led to the idea of re-implementing the classes in Java I/O that deals with file I/O.

A new library was therefore created and named `j2ee.io`. This library gives the web application authors, using the J2EE Public Service to deploy their applications, a way to access files stored on a specific location at DFS. The only necessary steps are to include a library file and a properties file giving credentials for accessing DFS through WebDAV and HTTPS in their web application.

This final `j2ee.io` library provides a secure and transparent way to access files and write new content back to DFS. Confidentiality of the files is taken care of by file system account access rights and Java fine-grained access control already existing in the J2EE Public Service.

However, it is important to note that file writing using the `j2ee.io` library has a problem; No file locking is implemented, which can lead to lost updates if used throughout a web application that has many users. Hence, this feature must be used with great care.

Bibliography

[1] The CERN name. http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/WhatIsCERN/CERNName/CERNName-en.html.

[2] What is CERN? http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/WhatIsCERN/WhatIsCERN-en.html.

[3] Who works there? http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/WhoWorksThere/WhoWorks-en.html.

[4] What are CERN's greatest achievements: The World Wide Web. http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/Achievements/WorldWideWeb/WWW-en.html.

[5] Wikipedia, the free Encyclopedia: World Wide Web. http://en.wikipedia.org/wiki/World_Wide_Web.

[6] What are CERN's greatest achievements: Nobel Prizes. http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/Achievements/NobelPrizes/NobelPrizes-en.html.

[7] What are CERN's greatest achievements: Colliding beams. http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/Achievements/CollidingBeams/CollidingBeams-en.html.

[8] Frequently Asked Questions: The LHC, How much does it cost? http://public.web.cern.ch/public/Content/Chapters/AskAnExpert/LHC-en.html.

[9] LHC Machine Outreach: Beam. http://lhc-machine-outreach.web.cern.ch/lhc-machine-outreach/beam.htm.

[10] The LHC experiments. http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/CERNFuture/LHCExperiments/LHCExperiments-en.html.

[11] Ben Berger. Deconstruction: Large Hadron Collider. *Symmetry Magazine*, 03, August 2006. http://www.symmetrymagazine.org/cms/?pid=1000364.

[12] How does the LHC work? http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/CERNFuture/HowLHC/HowLHC-en.html.

[13] What's next at CERN: Why the LHC? http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/CERNFuture/WhyLHC/WhyLHC-en.html.

[14] Worldwide LHC Computing Grid. http://lcg.web.cern.ch/LCG/.

[15] LCG Project Overview. http://lcg.web.cern.ch/LCG/overview.html.

[16] EGEE. http://www.eu-egee.org/.

[17] CERN Openlab. http://proj-openlab-datagrid-public.web.cern.ch/proj-openlab-datagrid-public/.

[18] The Technical Student Programme in Engineering, Computing and Applied Science. http://humanresources.web.cern.ch/humanresources/external/recruitment/students/tech.asp.

[19] Database and Engineering Services Group (IT-DES). http://it-des.web.cern.ch/IT-DES/.

[20] Database Infrastructure Services Section. http://it-des.web.cern.ch/IT-DES/DIS/.

[21] J2EE Public Service: What is J2EE Public Service? http://j2ee-public-service.web.cern.ch/j2ee-public-service/.

[22] CERN. *Cluster Architecture for Java Web Hosting at CERN*, February 2006. Paper from the CHEP 2006 conference in Mumbai, http://j2ee-public-service.web.cern.ch/j2ee-public-service/downloads/CHEP_2006_Paper_Cluster_architecture_for_java_web_hosting_at_CERN.pdf.

[23] Monitoring of the J2EE Public Service. http://jpsmanager.cern.ch/jps/central/showMonitoringInfo.do.

[24] J2EE Public Service Mandate. http://j2ee-public-service.web.cern.ch/j2ee-public-service/mandate.html.

[25] Michal Kwiatek. TWiki CERN internal: J2EEPS - How it works. https://twiki.cern.ch/twiki/bin/viewauth/DbaServices/HowItWorks.

[26] Michal Kwiatek. TWiki CERN internal: J2EEPS - Architecture. https://twiki.cern.ch/twiki/bin/viewauth/DbaServices/ServiceArchitecture.

[27] Michal Kwiatek. TWiki CERN internal: J2EEPS - JPSManager Software Architecture. https://twiki.cern.ch/twiki/bin/viewauth/DbaServices/JPSManagerSoftwareArchitecture.

[28] R. Fielding et al. *HyperText Transfer Protocol – HTTP/1.1*. W3C Network Working Group, June 1999.

[29] Wikipedia, the free Encyclopedia: Hypertext Transfer Protocol. http://en.wikipedia.org/wiki/HTTP.

[30] HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/.

[31] Tim Berners Lee et al. *Uniform Resource Identifier (URI): Generic Syntax*. W3C Network Working Group, January 2005.

[32] University of Southern California Information Sciences Institute. *Transmission Control Protocol*. 4676 Admiralty Way, Marina del Rey, California 90291, September 1981.

[33] University of Southern California Information Sciences Institute. *Internet Protocol*. 4676 Admiralty Way, Marina del Rey, California 90291, September 1981.

[34] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Fourth Edition edition, 2003.

[35] T. Dierks et al. *The Transport Layer Security (TLS) Protocol Version 1.1*. W3C Networking Group, April 2006.

[36] Y. Goland et al. *HTTP Extensions for Distributed Authoring – WebDAV*. W3C Networking Group, February 1999.

[37] E. James Whitehead Jr. Collaborative Authoring on the Web: Introducing WebDAV, October 1998. `http://www.asis.org/Bulletin/Oct-98/webdav.html`.

[38] E. James Whitehead Jr. and Meredith Wiggins. WEBDAV: IETF Standard for Collaborative Authoring on the Web. *Internet Computing*, 2(5), September - October 1998. `http://www.ics.uci.edu/pub/ietf/webdav/intro/webdav_intro.pdf`.

[39] Wikipedia, the free Encyclopedia: Denial-of-service attack. `http://en.wikipedia.org/wiki/Denial_of_service`.

[40] Joseph Bergin. Understanding Java I/O Facilities. `http://csis.pace.edu/~bergin/sol/java/gui/javaio.html`.

[41] I/O Streams. `http://java.sun.com/docs/books/tutorial/essential/io/streams.html`.

[42] Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison Wesley, August 2000. `http://java.sun.com/developer/Books/performance/ch04.pdf`, page 52.

[43] Buffered Streams. `http://java.sun.com/docs/books/tutorial/essential/io/buffers.html`.

[44] File Objects. `http://java.sun.com/docs/books/tutorial/essential/io/file.html`.

[45] Bruce Eckel. *Thinking in Java, 3rd Edition*. Prentice-Hall, December 2002. `http://www.javaolympus.com/thinking/TIJ314.htm`.

[46] The Apache Jakarta Project. `http://jakarta.apache.org/`.

[47] The Apache Software Foundation. `http://www.apache.org/foundation/`.

[48] Apache HTTP Server Project. `http://httpd.apache.org/`.

[49] Apache Tomcat. `http://tomcat.apache.org/`.

[50] Frequent Questions about Apache Licensing: I'm not a lawyer. what does it all MEAN? http://www.apache.org/foundation/licence-FAQ.html.

[51] Slide. http://jakarta.apache.org/slide/index.html.

[52] Slide 2.2pre1 WebDAV Client Javadoc. http://jakarta.apache.org/slide/clientjavadoc/index.html.

[53] Javadoc for the WebdavResource class. http://jakarta.apache.org/slide/clientjavadoc/org/apache/webdav/lib/WebdavResource.html.

[54] Jakarta Commons HTTPClient. http://jakarta.apache.org/commons/httpclient/.

[55] Commons Virtual File System. http://jakarta.apache.org/commons/vfs/.

[56] Supported File Systems. http://jakarta.apache.org/commons/vfs/filesystems.html.

[57] Welcome to WebDAV Resources. http://www.webdav.org/.

[58] Wikipedia, the free Encyclopedia: Proof of concept. http://en.wikipedia.org/wiki/Proof_of_concept.

[59] Gary D. Gregory. A simple command-line shell for performing file operations. http://svn.apache.org/repos/asf/jakarta/commons/proper/vfs/tags/vfs-1.0/examples/src/main/java/org/apache/commons/vfs/example/Shell.java.

[60] Javadoc for the FileSystemManager class. http://jakarta.apache.org/commons/vfs/apidocs/org/apache/commons/vfs/FileSystemManager.html.

[61] Javadoc for the FileObject class. http://jakarta.apache.org/commons/vfs/apidocs/org/apache/commons/vfs/FileObject.html.

[62] JDOM. http://www.jdom.org/.

[63] Javadoc for the File class. http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html.

[64] ASF Bugzilla Bug 32886: client webdav lib doesn't return child collections using listwebdavresources. http://issues.apache.org/bugzilla/show_bug.cgi?id=32886, Comment 16.

Appendices

Appendix A

Overview of the java.io package

Interface name	Description
Closeable	A source or destination of data that can be closed
DataInput	Provides reading of bytes from a binary stream and reconstruction from these data in any of the Java primitive types
DataOutput	Provides converting of data from any of the Java primitive types to a series of bytes and writing these to a binary stream
Externalizable	Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances
FileFilter	A filter for abstract pathnames
FileNameFilter	For filtering of filenames
Flushable	A destination of data that can be flushed
ObjectInput	Extends DataInput to include reading of objects
ObjectInputValidation	Callback interface to provide validation of objects within a graph
ObjectOutput	Extends DataOutput to include writing of objects
ObjectStreamConstants	Constants written into the Object Serialization
Serializable	Enables serializability of the class that implements it

Table A.1: Overview of interfaces in the `java.io` package

Class name	Description
BufferedInputStream	Adds functionality to another input stream by the ability to buffer the input and support the `mark` and `reset` methods
BufferedOutputStream	Implements a buffered input stream
BufferedReader	Reads text from a character-input stream, buffering characters for efficient reading
BufferedWriter	Writes text to a character input stream, buffering characters for efficient writing
ByteArrayInputStream	Contains an internal buffer that contains bytes that may be read from the stream
ByteArrayOutputStream	Implements an output stream where the data is written into a byte array
CharArrayReader	Implements a character buffer that can be used as a character input stream
CharArrayWriter	Implements a character buffer that can be used as a writer
DataInputStream	Lets an application read primitive Java data types from an underlying input stream in a machine-independent way
DataOutputStream	Lets an application write primitive Java data types to an output stream in a portable way
File	An abstract representation of file and directory pathnames
FileInputStream	Obtains input bytes from a file in a file system
FileOutputStream	An output stream for writing data to a `File` or a `FileDescriptor`
FileDescriptor	Serves as an opaque handle to the underlying machine-specific structure representing an open file, an open socket or another source or sink of bytes
FilePermission	Represents access to a file or directory
FileReader	Class for reading character files
FileWriter	Class for writing character files
FilterInputStream	Contains some other input stream used as basis for its data, and possibly transforms the data or provides additional functionality

FilterOuputStream	Superclass of all classes that filters output streams
FilterReader	Abstract class for reading filtered character streams
FilterWriter	Abstract class for writing filtered character streams
InputStream	Abstract class who is the superclass of all classes representing an input stream of bytes
InputStreamReader	Bridge from byte streams to character streams
LineNumberInputStream	`Deprecated`
LineNumberReader	A buffered character input stream that keeps track of line numbers
ObjectInputStream	Deserializes primitive data and objects previously written using an ObjectOutputStream
ObjectInputStream.GetField	Provides access to the persistent fields read from the input stream
ObjectOutputStream	Writes primitive data types and graphs of Java objects to an OutputStream
ObjectOutputStream.PutField	Provides programmatic access to the persistent fields to be written to ObjectOutput
ObjectStreamClass	Serialization's descriptor for classes
ObjectStreamField	A description of a Serializable field from a Serializable class
OutputStream	Abstract class which is the superclass of all classes representing an output stream of bytes
OutputStreamWriter	A bridge from character streams to byte streams
PipedInputStream	When a connection to a piped output stream is established, this class provides whatever data bytes are written to the piped output stream
PipedOutputStream	Can be connected to a piped input stream to create a communications pipe
PipedReader	Reads piped character input streams

PipedWriter	Writes piped character output streams
PrintStream	Adds the ability to print presentations of various data conveniently to other output streams
PrintWriter	Prints formatted representations of objects to a text output stream
PushbackInputStream	Adds the ability to "push back" or "unread" one byte to another input stream
PushbackReader	A character stream reader that allows characters to be pushed back into the stream
RandomAccessFile	Instances of this class support both reading and writing to a random access file
Reader	Abstract class for reading character streams
SequenceInputStream	Represents the logical concatenation of other input streams
SerializablePermission	A class for Serializable permissions
StreamTokenizer	Takes an input stream and parses it into "tokens", allowing the tokens to be read one by one
StringBufferInputStream	`Deprecated`
StringReader	A character stream whose source is a string
StringWriter	A character stream that collects its output in a string buffer, that later can be used to construct a string
Writer	Abstract class for writing to character streams

Table A.2: Overview of classes in the `java.io` package

Exception name	Description
CharConversionException	Base class for character conversion exceptions
EOFException	Signals that an end of a file or end of a stream has been reached unexpectedly
FileNotFoundException	Signals that an attempt to open the file given by a specific pathname has failed
InterruptedIOException	Signals that an I/O operation has been interrupted
InvalidClassException	Thrown when the Serialization runtime detects some specific problems with a class
InvalidObjectException	Indicates that one or more deserialized objects failed validation tests
IOException	Signals that an I/O error has occurred
NotActiveException	Thrown when serialization or deserialization is not active
NotSerializableException	Thrown when an instance is required to have a Serializable interface
ObjectStreamException	Superclass of all exceptions specific to Object Stream classes
OptionalDataException	Indicating the failure of an object read operation due to unread primitive data, or the end of data belonging to a serialized object in the stream
StreamCorruptedException	Thrown when control information that was read from an object stream violates internal consistency checks
SyncFailedException	Signals that a sync operation has failed
UnsupportedCharacterEncoding	Signals that an unsupported character encoding is being used
UTFDataFormatException	Signals that a malformed string in modified UTF-8 format has been read in a data input stream or by a class that implements the data input interface
WriteAbortedException	Signals that one of the ObjectStreamExceptions was thrown during a write operation

Table A.3: Overview of exceptions in the `java.io` package

Appendix B

Patching procedure for the Slide WebDAV library

Because of a bug in the library that was used where directories are not retrieved correctly using the `listWebdavResources()` in the `org.apache.webdav.lib.WebdavResource` class, it is necessary to replace the `setWebdavProperties()` method in the class `org.apache.webdav.lib.WebdavResource` with the code shown below[64]:

```
/**
 * Set WebDAV properties following to the given http URL.
 * This method is fundamental for getting information of a collection.
 *
 * @param responses An enumeration over {@link ResponseEntity} items, one
 * for each resource for which information was returned via PROPFIND.
 *
 * @exception HttpException
 * @exception IOException The socket error with a server.
 */

protected void setWebdavProperties(Enumeration responses)
        throws HttpException, IOException {

        // Make the resources in the collection empty.
        childResources.removeAll();
        while (responses.hasMoreElements()) {

            ResponseEntity response = (ResponseEntity) responses.nextElement();

            boolean itself = false;
            String href = response.getHref();
            if (!href.startsWith("/"))
                href = URIUtil.getPath(href);
            href = decodeMarks(href);

            /*
            Decode URIs to common (unescaped) format for comparison
            as HttpClient.URI.setPath() doesn't escape $ and : chars.
            */
            String httpURLPath = httpURL.getPath();
            String escapedHref = URIUtil.decode(href);

            // Normalize them to both have trailing slashes
            // if they differ by one in length.
            int lenDiff = escapedHref.length() - httpURLPath.length();
            int compareLen = 0;

            if ( lenDiff == -1 && !escapedHref.endsWith("/")) {
                compareLen = escapedHref.length();
```

```
        lenDiff = 0;
    }
    else if ( lenDiff == 1 && !httpURLPath.endsWith("/")) {
        compareLen = httpURLPath.length();
        lenDiff = 0;
    }

    // if they are the same length then compare them.
    if (lenDiff == 0) {
        if ((compareLen == 0 && httpURLPath.equals(escapedHref))
            || httpURLPath.regionMatches(0, escapedHref, 0, compareLen))
        {
            // escaped href and http path are the same
            // Set the status code for this resource.
            if (response.getStatusCode() > 0)
                setStatusCode(response.getStatusCode());
            setExistence(true);
            itself = true;
        }
    }

    // Get to know each resource.
    WebdavResource workingResource = null;
    if (itself) {
        workingResource = this;
    }
    else {
        workingResource = createWebdavResource(client);
        workingResource.setDebug(debug);
    }

    // clear the current lock set
    workingResource.setLockDiscovery(null);

    // Process the resource's properties
    Enumeration properties = response.getProperties();
    while (properties.hasMoreElements()) {

        Property property = (Property) properties.nextElement();

        // Checking WebDAV properties
        workingResource.processProperty(property);
    }

    String displayName = workingResource.getDisplayName();

    if (displayName == null || displayName.trim().equals("")) {
        displayName = getName(href);
    }

    /** BUGGY CODE
    if (!itself) {
        String myURI = httpURL.getEscapedURI();
        char[] childURI = (myURI + (myURI.endsWith("/") ? "" : "/")
                           + URIUtil.getName(href)).toCharArray();
        HttpURL childURL = httpURL instanceof HttpsURL
                           ? new HttpsURL(childURI)
                           : new HttpURL(childURI);
        childURL.setRawAuthority(httpURL.getRawAuthority());
        workingResource.setHttpURL(childURL, NOACTION, defaultDepth);
        workingResource.setExistence(true);
        workingResource.setOverwrite(getOverwrite());
    }
    */

    /** FIX ********/
    if (!itself) {
        String myURI = httpURL.getEscapedURI();

        /**
```

```
                    Checks if href contains trailing '/', and if so removes it.
                    This ensures URIUtil.getName does not return an empty
                    String when we don't want it to.

                    See http://issues.apache.org/bugzilla/show_bug.cgi?id=32886
                    for more information.
                    */
                   String fixedHref = href.endsWith("/") ?
                        href.substring(0, href.length() - 1) : href;

                   char[] childURI = (myURI + (myURI.endsWith("/") ? "" : "/")
                                      + URIUtil.getName(fixedHref)).toCharArray();

                    HttpURL childURL = httpURL instanceof HttpsURL
                                       ? new HttpsURL(childURI)
                                       : new HttpURL(childURI);
                    childURL.setRawAuthority(httpURL.getRawAuthority());
                    workingResource.setHttpURL(childURL, NOACTION, defaultDepth);
                    workingResource.setExistence(true);
                    workingResource.setOverwrite(getOverwrite());
               }
               /**************/

               workingResource.setDisplayName(displayName);

               if (!itself)
                    childResources.addResource(workingResource);
          }
     }
```

Appendix C

Overview of electronical documentation

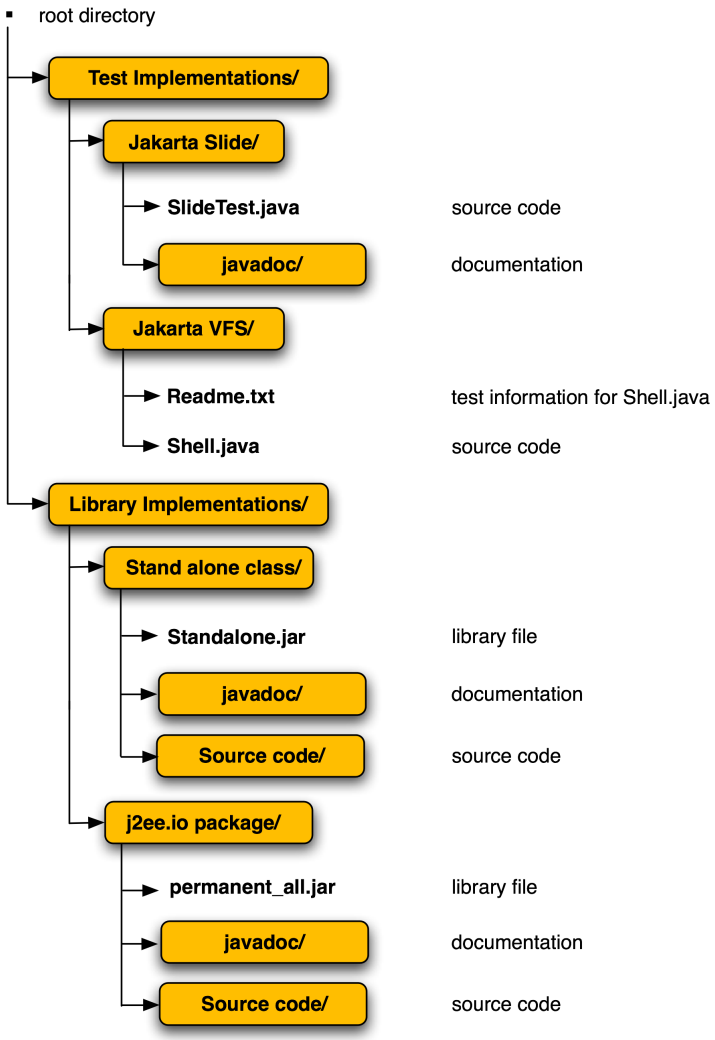

Figure C.1: Overview of the *documentation.zip* directory structure