Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2016,1
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

The Media Store 3 Case Study System

Misha Strittmatter*, Amine Kechaou

2016

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

*Institute for Program Structures and Data Organization (IPD)
Karlsruhe Institute of Technology (KIT)

Please note:
This Report has been published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de.

Contents

1 Introduction **4**
1.1 License . . . 4
1.2 Resources . . . 5

2 Architecture **6**

3 Implementation **8**
3.1 Setup . . . 8
3.1.1 Technical requirements . . . 8
3.1.2 Server setup . . . 8
3.1.3 Database setup . . . 8
3.1.4 File Location Configuration . . . 10
3.1.5 Deployment . . . 10
3.2 General Information . . . 11
3.3 Code Structure . . . 16
3.3.1 Structural overview . . . 16
3.3.2 The frontend: A Web application . . . 17
3.3.3 The backend: An EJB architecture . . . 17
3.3.4 Mediastore.basic . . . 18
3.4 Reconfiguration . . . 19
3.5 Intelligent Remote and Local EJB Calls . . . 22

4 PCM Model **24**
4.1 Description . . . 24
4.2 Audio Payload . . . 26
4.3 Hardware Setup . . . 26
4.4 Model Files . . . 27
4.5 Launches . . . 29
4.6 Modeling of the Cache Component . . . 29

5 Data **33**

Bibliography **36**

1 Introduction

This document serves as technical documentation of the third implementation of the Media Store case study system. It gives rationale behind some specifics within the implementation and the PCM model. The purpose of Media Store system is to serve as a performance case study system. Media Store is a multi-user web-based distributed file loader application for the sharing of audio files. It is used as a running example in the Palladio Book [1]. For more background information (e.g., domain requirements, rationale behind the functionality and architecture), please consult the book. Media Store is implemented on Java EE (Enterprise Edition). Components, which are realized by EJBs (Enterprise Java Beans), should be units of deployment. Therefore, in the version 1.0 of the implementation, all calls are remote to ensure free deployment. Later this behavior was improved, as using remote calls for locally deployed EJBs is inefficient. In version 1.2, a mechanism was implemented for an automatic dispatch of local and remote calls. Since version 1.1 a mechanism is in-place which enables runtime reconfiguration of the architecture. The PCM model of Media Store was built to reflect the performance of version 1.0 of the implementation. The model should match version 1.1 at least concerning the tendencies. However, as version 1.2 bring great performance improvement depending on deployment, we do cannot guarantee reasonable fit.

This report is structured as follows. After the license information, a brief overview is given about all resources available regarding Media Store and where they can be obtained online. Later, the implementation independent component architecture of Media Store is presented in Section 2. Section 3 describes the Java EE implementation of Media Store. The PCM performance model of Media Store and its calibration is explained in Section 4. The calibration and evaluation data is described in Section 5.

1.1 License


```
Media Store V3
Copyright (C) 2015 Software Design and Quality Group (SDQ), KIT, Germany

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
```

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

Full version of the GNU General Public License is at:
../Implementation/gpl-3.0.txt
or
http://www.gnu.org/licenses/

1.2 Resources

All information about Media Store can be found at the following SVN repository:
https://svnserver.informatik.kit.edu/i43/svn/code/CaseStudies/MediaStore3
The structure of the repository is as follows:

```
MediaStore3
├─branches
│ └─Nightly Model
├─tags
│ ├─Mediastore 1.0
│ ├─Mediastore 1.1
│ └─Mediastore 1.2
└─trunk
  ├─Data
  ├─Implementation
  ├─Model
  └─TechReport
```

The trunk folder contains the most up-to-date files. Keep in mind, that this code may be under development and may contain errors and may not be in sync with the model or the data. The tags folder contains tagged versions of the trunk or of subfolders of the trunk. Currently there are three versions of the implementation tagged: version 1.0 is the version which was measured for comparison with the model prediction. Version 1.1 features a new mechanism for runtime reconfiguration. Version 1.2 automatically performs local calls instead of remote ones where possible. The trunk contains the following subfolders:

Data Measurement and simulation data of Media Store (see Section 5).
Implementation The source code of Media Store (see Section 3).
Model The PCM model of Media Store (see Section 4).
TechReport Sources and compiled version of this document.

The branches folder features parallel development to the trunk. Currently it contains a version of the model that was migrated to the currently nightly version of the PCM.

2 Architecture

The architecture of Media Store is component-based. It is shown in Figure 2.1. It is divided into three layers: 1) The frontend simply consists of the GUI. 2) The business logic consists of several components that handle user management, audio management and processing. 3) The storage layer consists of components that handle database and file access, a database and a data storage for the audio files. Media Store provides two types of functionality: audio storage and user management. Users can register new accounts. Once registered, they can perform a log in, to gain access to the audio functionality. This includes uploading audio files, listing all stored audio files and downloading audio files. In the following, we will explain the responsibilities of the components of the architecture.

The GUI component is the graphical user interface of Media Store. It provides all the aforementioned functionality and keeps track of sessions. It is usually implemented as a web GUI. According to actions of the user, it creates requests which are sent into the system's backend.

The UserManagement component provides the business logic for the management of the information of the Media Stores users. The component serves register and login requests. Information, which is stored in the DB in an encrypted fashion, is decrypted here. It uses the UserDBAdapter component to get the needed information from the DB. UserDBAdapter builds and submits the required queries to the DB and returns the acquired information.

The MediaManagement serves upload, download and list request concerning the stored audio files. It forwards list and upload requests directly to the MediaAccess component. Download requests are sent via a separate download interface. At this interface, the architecture is variable. This is indicated by the big square brackets. The components in the brackets form a chain of responsibility by providing and requiring interfaces of the same type. There are multiple options of how to assemble this chain. The simplest option is not to have any components in the chain. Requests are then passed from MediaManagement directly to MediaAccess. A Watermarking component applies digital watermarks, which carrying information about the downloading user, onto the audio files upon download. A ReEncoder reencodes audio files in a specific quality, encoding or bit rate. A Cache stores commonly requested files (possibly in specific qualities). For requests resulting in a cache hit, this saves fetching and reencoding the file. In the chain, none, one or multiple of these components may be used. There are several constraints regarding the order of components in the chain, which are in part implementation specific. E.g. should Watermarking and Cache be used, Watermarking should be in front of Cache, as it is not meaningful to cache watermarked components. Should a Watermarking implementation require a reencoding of the audio file, a ReEncoder component is not necessary.

After a download request for a collection of audio files has returned to MediaManagement, it may send it to Packaging. If the request resulted in a set of audio files, the Packaging component

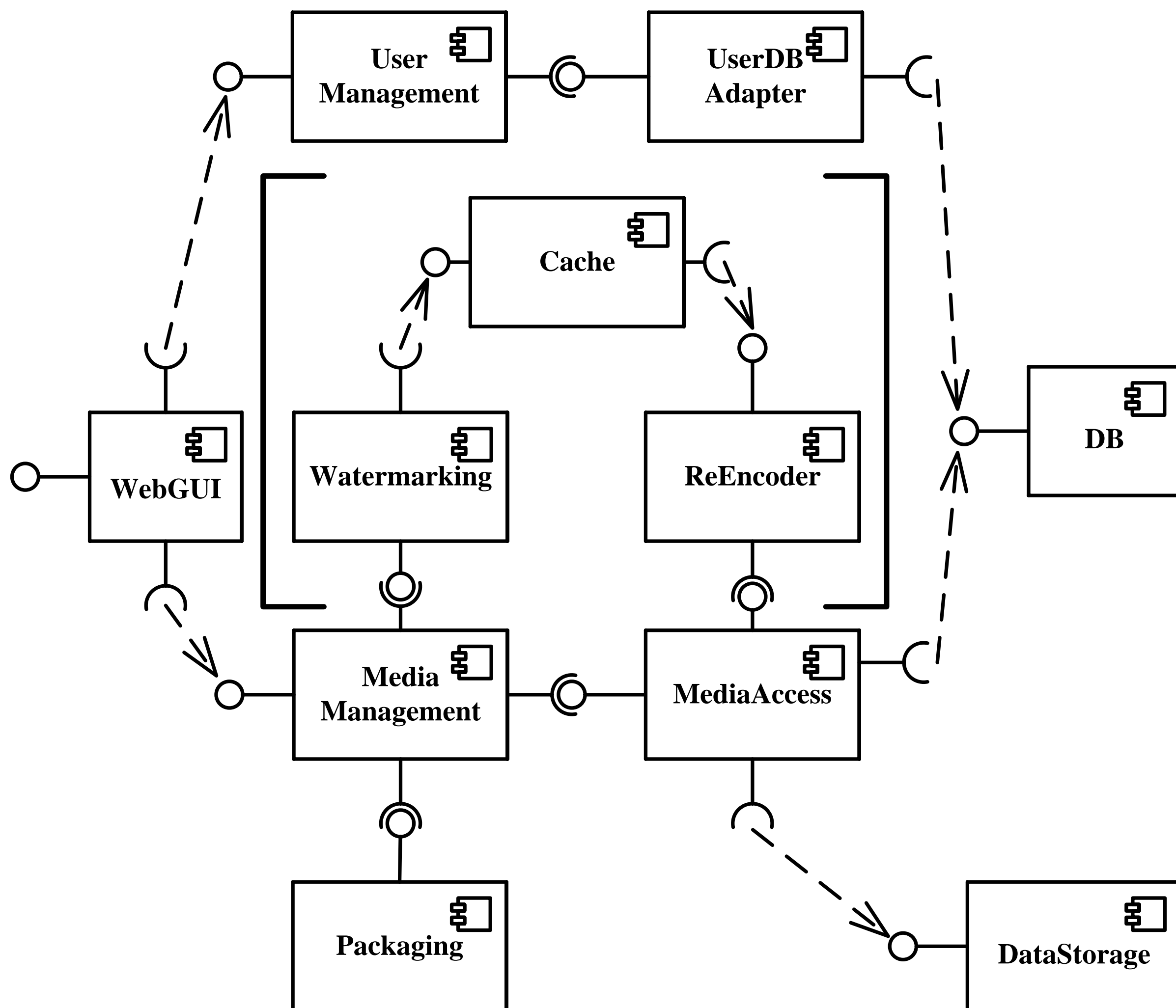

Figure 2.1: Media Store's Architecture

to create a single compressed archive.

MediaAccess is responsible for fetching information about audio files from the DB and the audio files themselves from DataStorage. DB and DataStorage are no components in the strict sense. However, they are shown in the architecture for illustrative purposes. DB is usually a DBMS (database management system) and DataStorage could be a directory directly on a file system or a network share.

3 Implementation

3.1 Setup

3.1.1 Technical requirements

- Java Development Kit 1.8[1]
- Eclipse IDE for Java EE Developers[2]
- Glassfish 4.1
- MySQL

3.1.2 Server setup

First, download GlassFish 4.1[3]. There is no installation required, all is needed is to unzip the file. After unzipping, the domain domain1 is created by default. There are several ways to manage domains.

in Eclipse

- Start by installing Glassfish Tools from the Eclipse marketplace.
- In the servers view, right click then click New > Server.
- Proceed by following the wizard instructions.

From the command line

The domain domain1 is located at /glassfish/domains/domain1. To start it, launch asadmin at /glassfish/bin then type the command "start-domain domain1". The command to stop is "stop-domain domain1". In order to created another domain, say domain2, type the command "create-domain domain2". To delete it, use the command "delete-domain domain2". To list all domains and there status, use the command "list-domains".

3.1.3 Database setup

There are various ways to install and manage a MySQL database. One of the easiest, is to install XAMPP, which is a bundle of an Apache Server, a MySQL database and interpreters

[1] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[2] https://eclipse.org/downloads/
[3] http://download.java.net/glassfish/4.1/release/glassfish-4.1.zip

for PHP and Perl. It comes also with other modules, such as phpMyAdmin, which is basically a GUI for MySQL. After installing XAMPP, launch phpMyAdmin and create a new database named mediastore. Then, create two tables audio and user. The SQL code in Listing 1 can be used for that purpose.

```
CREATE TABLE IF NOT EXISTS 'audio' (
  'ID' int(11) NOT NULL,
  'ALBUM' varchar(255) DEFAULT NULL,
  'ARTIST' varchar(255) NOT NULL,
  'BITRATE' int(11) NOT NULL,
  'GENRE' varchar(255) DEFAULT NULL,
  'RELEASEYEAR' int(11) DEFAULT NULL,
  'TITLE' varchar(255) NOT NULL,
  'USERID' bigint(20) NOT NULL,
  PRIMARY KEY ('ID')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS 'user' (
  'ID' bigint(20) NOT NULL AUTO_INCREMENT,
  'EMAIL' varchar(255) NOT NULL,
  'FIRSTNAME' varchar(255) NOT NULL,
  'LASTNAME' varchar(255) NOT NULL,
  'PASSWORD' varchar(255) NOT NULL,
  PRIMARY KEY ('ID')
);
```

Listing 1: Table Creation SQL Query

The next step is to configure the JDBC Resource on the default domain domain1:

1. Stop domain1.
2. Download the MySQL connector [4] and place it in /glassfish/domains/domain1/lib/
3. Start domain1 and open the admin console.[5]
4. Create a new JDBC connection pool. Set type to java.sql.DataSource and DataBase driver vendor to MySQL and specify the following properties as mentioned :

   Servername: localhost[6]
   DatabaseName: mediastore
   Port: 3306
   URL and url: jdbc:mysql://localhost:3306/mediastore
   user and password: Use the same used to create the database

[4] http://central.maven.org/maven2/mysql/mysql-connector-java/5.1.36
[5] Go to http://localhost:4848
[6] can either point to a remote or local server

5. Save then create a new JDBC Resource using the JNDI Name jdbc/Mediastore and the new connection pool.

3.1.4 File Location Configuration

Media Store has two location where it stores its audio files. If these folders do not exist, Media Store attempts to create them.

One of the locations is the final storage of the audio files. There the MediaAccess component stores uploaded files and retrieves files which are requested for download. In the Architecture it is depicted as the DataStorage component. The other location is used for temporary storage of files while processing. It is used by several components.

In the version 1.2, the file configuration is managed through an external file GlobalConstantsContainer.properties, which can be found at the root of edu.kit.ipd.sdq.mediastore.basic. The file must be copied to the config folder of the running domain[7]. It may be then modified on the fly without having to redeploy the application. The class GlobalConstantsContainer provides the methods getTempDirPath, getFileDir and getCacheCapacity which retrieve the needed properties from the configuration file by calling the private method getProperty(String prop). If it's the first time the configuration is to be loaded, or if the file has been modified, this method makes a call to loadProperties to load the configuration and then returns the desired property.

Back in versions 1.0 and 1.1 of Media Store, the two file locations can only be reconfigured by modifying constants in the source code. The constants are located in the project mediastore.basic in the class edu.kit.ipd.sdq.mediastore.basic.config.GlobalConstantsContainer. The path to the final audio storage is delivered by the static function getFileDir. Depending on the operating system, the constants FILE_DIR_WIN and FILE_DIR_LINUX hold the file location. Their default values are respectively C:\\audios\\ and /home/audios. The path to the temporary storage is defined by the constant TEMP_DIR_PATH. Its default value is C:\\temp\\.

3.1.5 Deployment

There are two ways to deploy Media Store.

Automatic Deploy

- Stop the target domain if it is already running.
- In the server view right-click on the target domain, then select Add and Remove. (Figure 3.1)
- Click on "Add all" then "Finish". (Figure 3.2)

[7] <glassfish install dir>/glassfish/domains/<domain>/config

Manual Deploy

- Right-click on the Enterprise Archive (EAR) (Figure 3.3) or the web project to be deployed and choose "Export"
- To deploy an EAR choose Java EE > EAR file (Figure 3.4). In case of a web project, choose Web > War file.
- Save the file into /glassfish/domains/domain1/autodeploy.

In addition to deploying the EARs, it is required up from Media Store 1.1 to copy the file present at mediastore.basic to the config folder of the running domain. See Section 3.4 for more information.

3.2 General Information

The purpose of Media Store implementation is not to be an application designed for use by humans, but to possess architectural and performance features like such an application. Using it as a subject of investigation, the capabilities of the Palladio Approach can be demonstrated. Thus, it is designed to adhere to a set of special requirements: component-based architecture, free individual deployment of each component, up-to-date Java EE technology, presence of scaling resource demands.

However, the free individual deployment of components has negative implications on the performance of the implementation, if EJBs are deployed and have to communicate within the same application server. Thus, since version 1.2, local calls are automatically performed instead of remote ones when possible. Using this mechanism it is possible to package multiple EJBs into one EAR and have them communicate locally. This solution profits from the flexibility of free deployment and the performance of local calls. It is described in section 3.5. Another feature is the runtime rewiring of EJBs. It was implemented in version 1.1 and is documented in section 3.4. As long as EJBs are already deployed, it is possible to inject provided interfaces into compatible provided interfaces.

The implementation adheres to a component to software artifact mapping. Figure 3.5 shows a possible configuration of the component-based architecture of Media Store, as specified in the PCM model. Basic components that are marked by the EJB stereotype map to stateless session beans. Provided interfaces (roles to be exact) map to implemented business interfaces of EJBs. Required interfaces map to business interfaces which appear as attributes of EJBs. Connectors can be regarded as the data, which is provided for the JNDI lookup, which retrieves the required EJB. Exceptions to this mapping are: the DB, DataStorage, WebGUI components and their interfaces. The DB component is an arbitrary JPA compatible DBMS. The DataStorage represents a location on a file system, which may be local or on a remote file share. The WebGUI is mainly JSF but uses an empty EJB (Facade) to dispatch its calls into the backend.

For some processing Media Store sometimes swaps the content of an audio file from memory to a file and vice versa. Also, the location of the file may switch if a remotely called component makes a local call or vice versa. To mitigate the effort needed to read a file from disc or write it to disc, a RAM disc tool can be used. The location of the temporary files can be changed in the

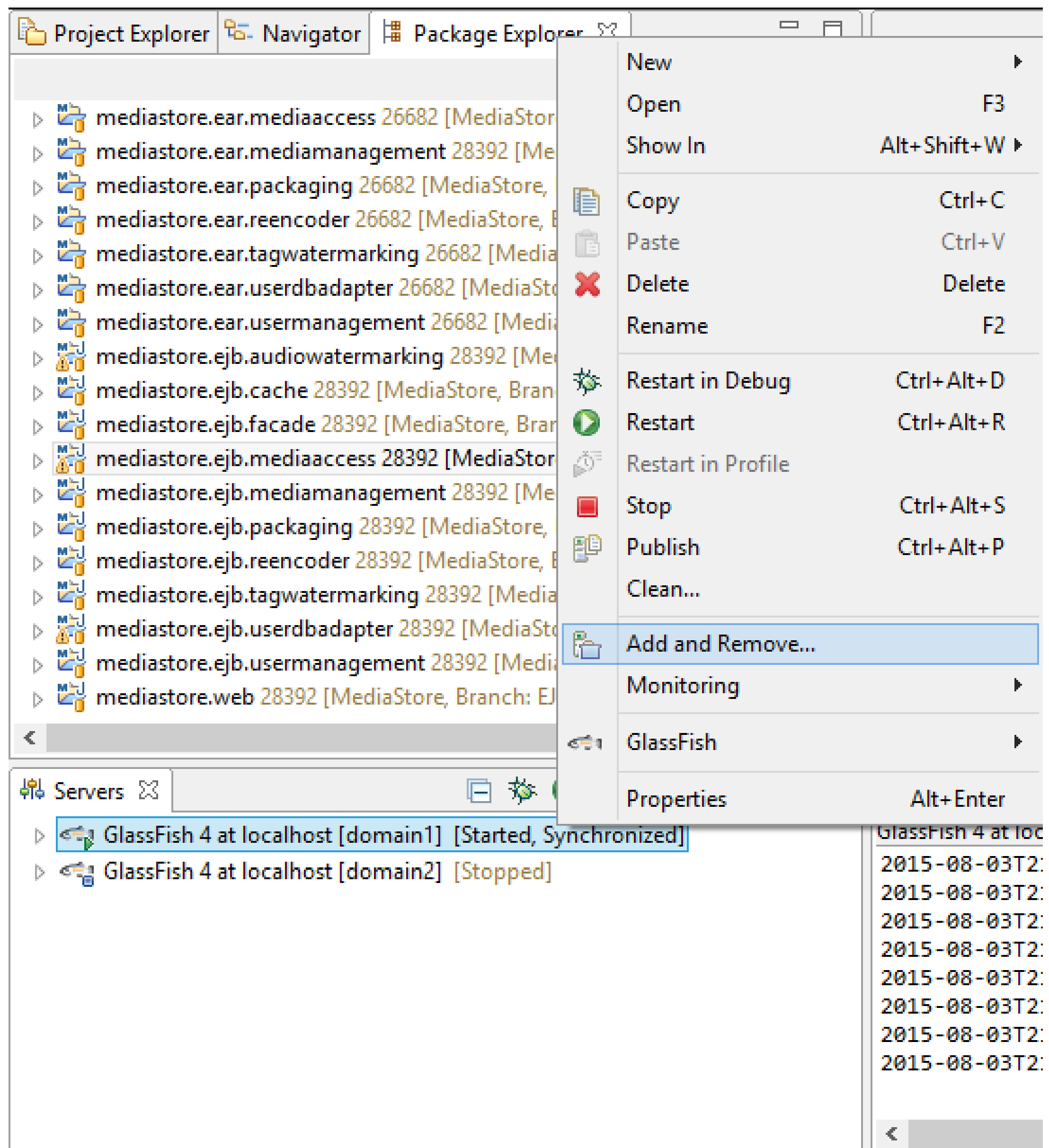

Figure 3.1: Server view

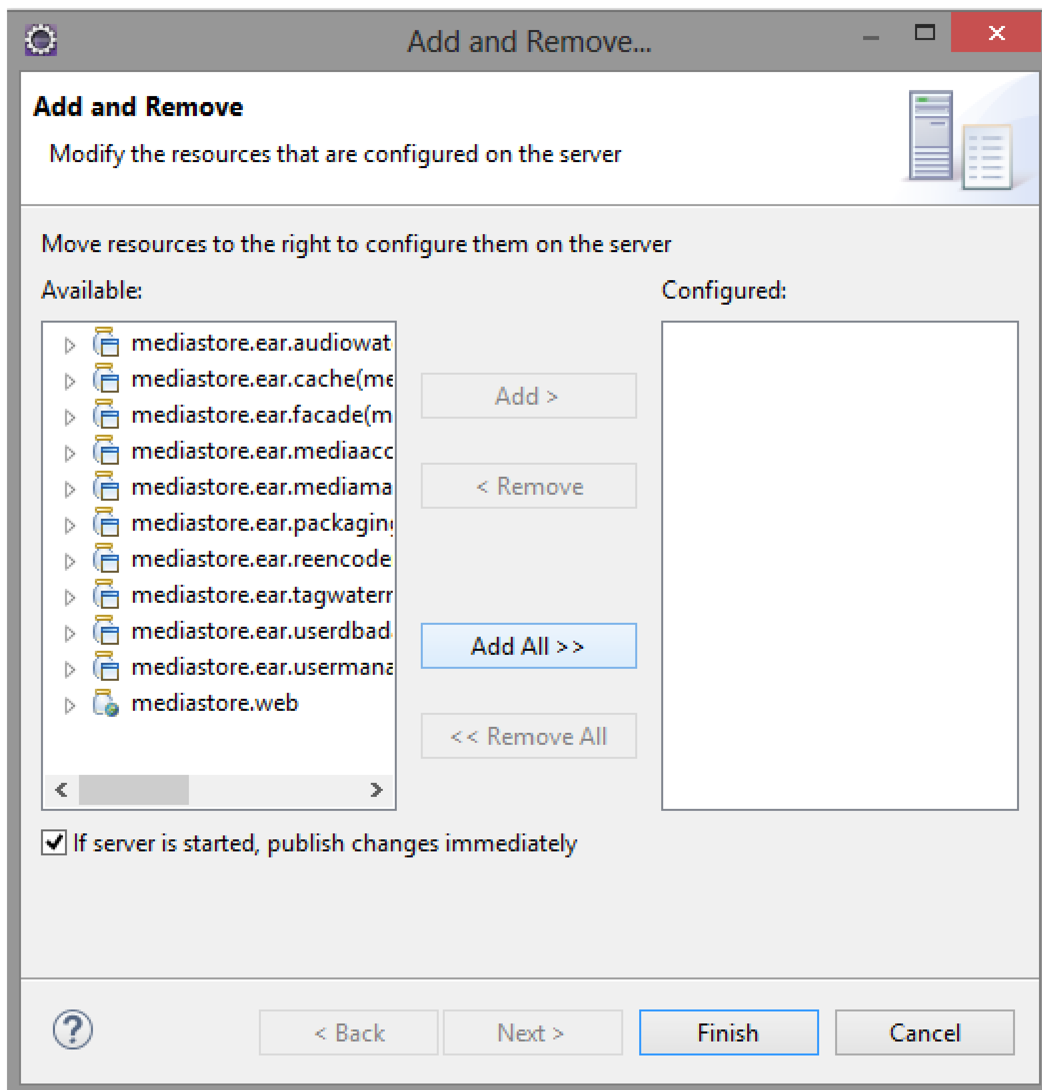

Figure 3.2: Add and Remove

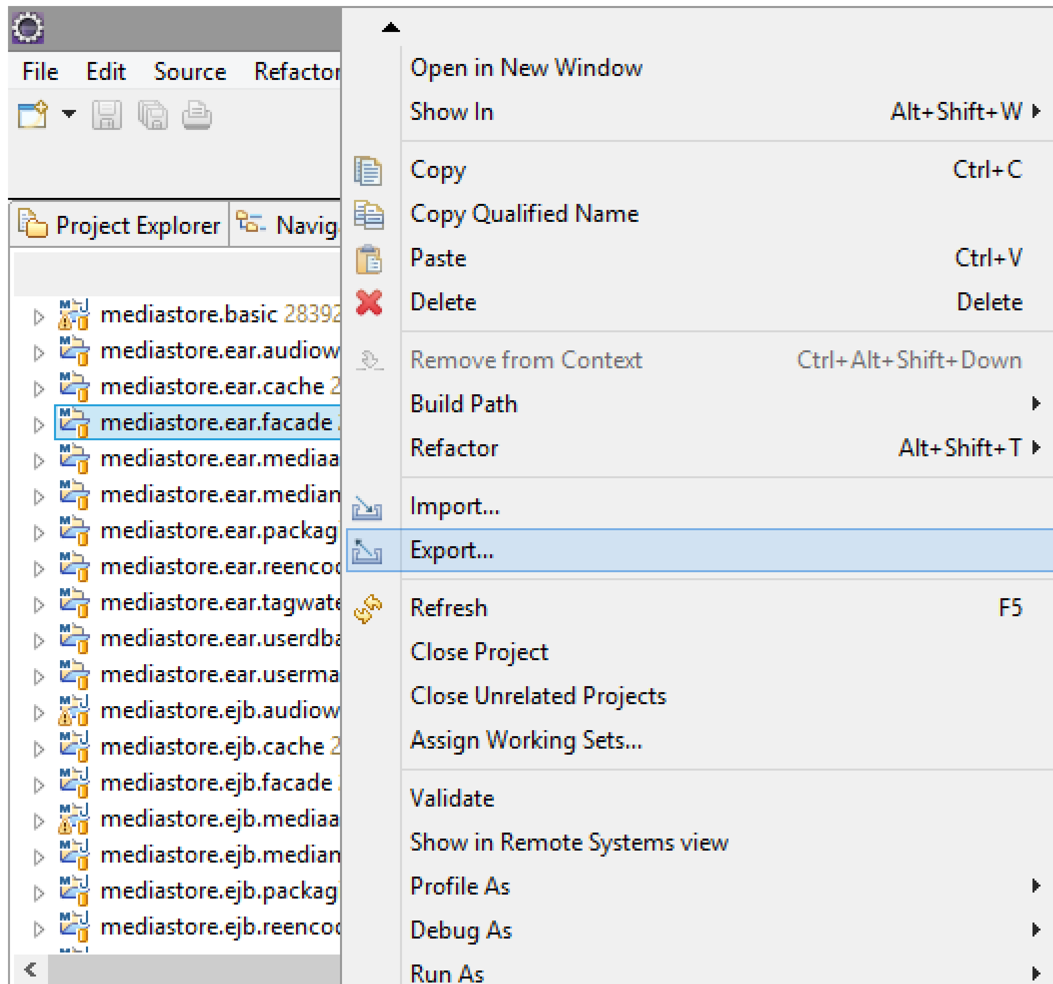

Figure 3.3: Exporting an EAR

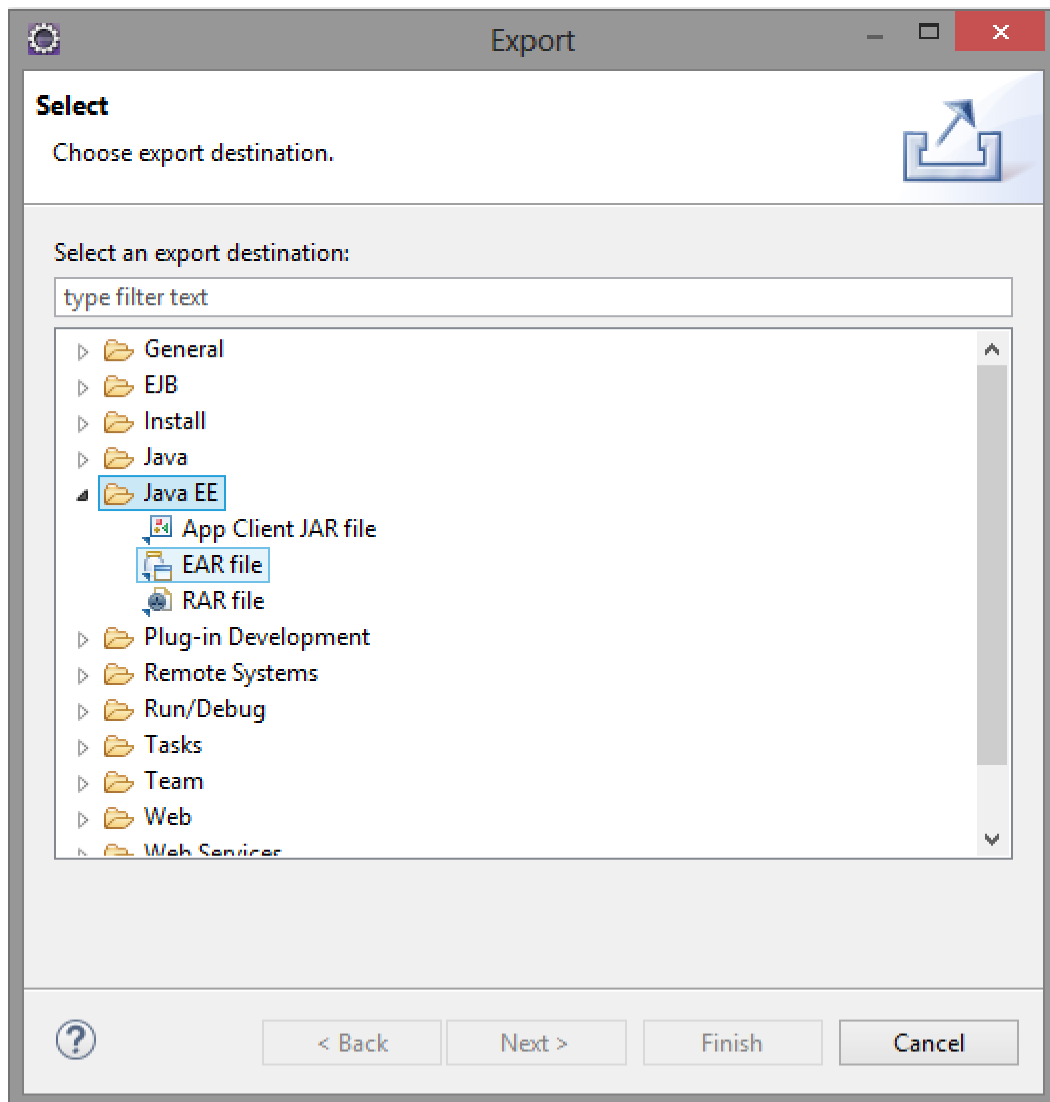

Figure 3.4: Export window

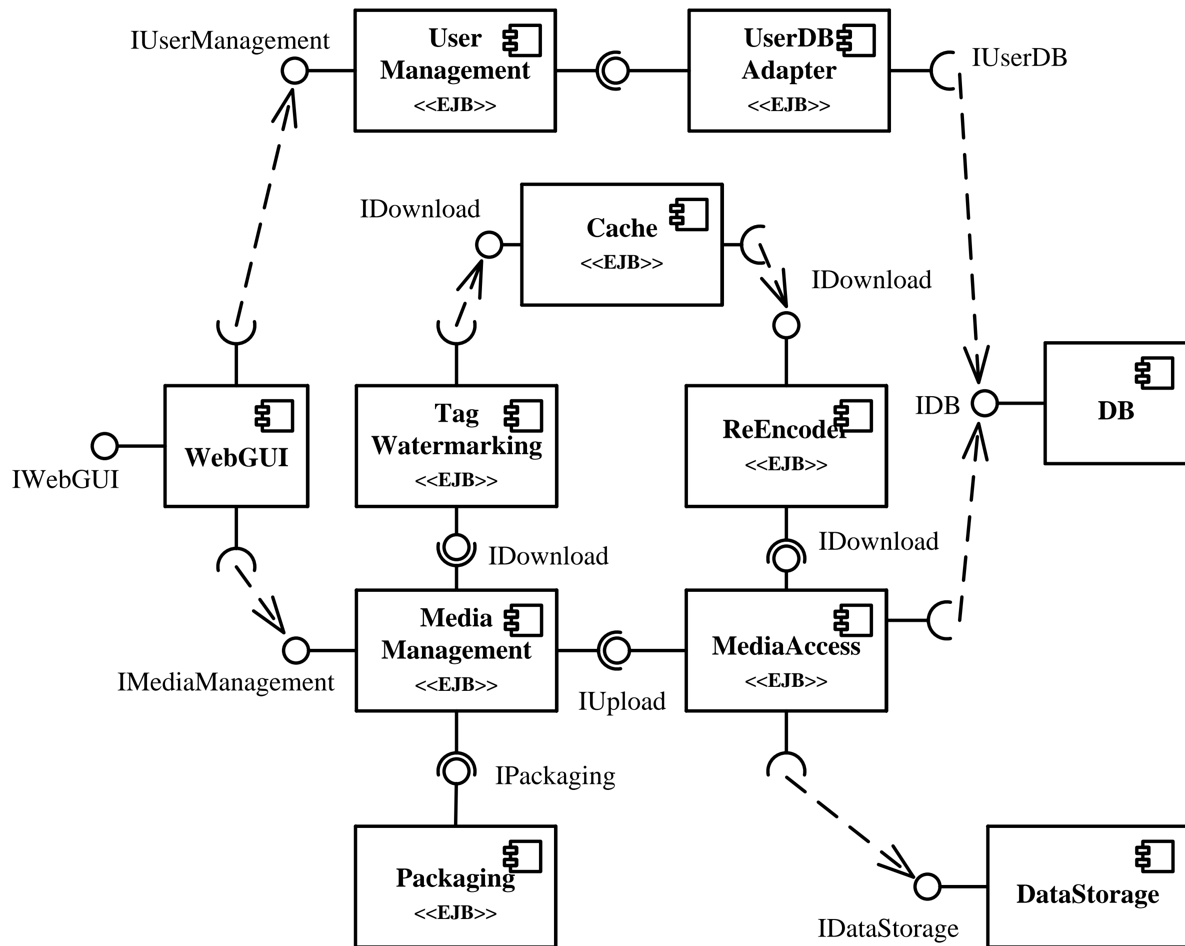

Figure 3.5: Possible Configuration of Media Store Architecture

source code. Thus, in the current version, this has to be done prior to deployment. The storage location of the audio files can also be changed in the same way. It can be located on a local hard disc or even on a remote file share. How the file locations can be configured is explained in subsection 3.1.4.

3.3 Code Structure

3.3.1 Structural overview

Apart from the Web Frontend, each component is realized by an Enterprise JavaBean (EJB) and packaged inside an Enterprise Archive (EAR). For each one of those are assigned two projects: An EJB Project and an EAR Project. However, if two or more EJBs should be deployed locally, a new EAR Project should be created to contain them all.

In addition to those, two more projects complete Media Store architecture: a Dynamic Web Project for the Web Frontend, and a simple java project called mediastore.basic, which holds common utilities and aspects shared between all other components. This will be covered more in detail in section 3.3.

Maven dependencies

All projects possess a Maven nature and therefore the dependencies between components are managed through Maven. For example, an EJB can be added to a EAR by simply adding the right dependency to its pom.xml file (See Listing 2). It is also possible to package multiple EJBs within the same EAR, which makes these EJBs local to one another. This brings some advantages as covered in Section 3.5.

```
1 <dependency>
2                   <groupId>mediastore.ejb.facade</groupId>
3                   <artifactId>mediastore.ejb.facade</artifactId>
4                   <version>1.0</version>
5                   <type>ejb</type>
6 </dependency>
```

Listing 2: Adding the Facade EJB to an EAR

3.3.2 The frontend: A Web application

The web frontend is implemented by mediastore.web as a Dynamic Web Project using the framework JavaServer Faces (JSF). The Graphical User Interface (GUI) is implemented in the HTML files index.xhtml, login.xhtml and register.xhtml using of the PrimeFaces framework. The business logic part of the Web component is realized by the Managed Beans in the package edu.kit.ipd.sdq.mediastore.web.beans. Those eventually make use of the utility classes at the package edu.kit.ipd.sdq.mediastore.web.utils

Other than that, the class MainFilter at the package edu.kit.ipd.sdq.mediastore.web.filters serves for redirecting and URL-renaming purposes. The use of this filter is specified in the configuration file mediastore.web/WebContent/WEB-INF/web.xml.

3.3.3 The backend: An EJB architecture

The business logic of the backend is realized by EJBs. More particularly, each component is implemented by at least one stateless Session Bean.

- The Facade component serves as an intermediary between the web frontend and the other components. Actions involving the user system, e.g. login, are redirected to the UserManagment component and actions involving audio files, e.g. download, are redirected to the MediaManagement component. It contains one Session Bean FacadeImpl, which implements the interfaces IFacadeLocal and IFacadeRemote.

- UserManagement is the component implementing the user system. It has one Session Bean UserManagementImpl, which implements the interfaces IUserManagementLocal and IUserManagementRemote. It also includes the class SecurityUtil which is a utility class for encryption purposes.
- UserDBAdapter is responsible for the interaction between the application and the user table in the database. It contains two Session Beans : DbManager, which offers the primitive functions of the interaction with the user table, and UserDBAdapterImpl wich serves as an entry point from the UserManagement component to the database. In addition to that, the UserDBAdapter component makes use of the Java Persistence API (JPA) and includes an entity class User, which represents an entry in the user table inside the database.
- The MediaManagement component coordinates with the other components upon a download or upload action. It is implemented in the Session Bean MediaManagementImpl, which implements the interfaces IMediaManagementLocal and IMediaManagementRemote
- MediaAccess is the component in charge of saving files on the storage device upon an upload, or retrieving those on download. This is done by the class MediaAccessImpl. The interaction with the database is done by the Session Bean DbManager. For this purpose, the entity class Audio represents an entry in the audio table of the database.
- AudioWatermarking applies a watermark on an audio file by adding an inaudible signal to it. It is implemented by the Session Bean AudioWatermarkingImpl and makes use of a Wave File API, implemented in the classes WavFile and WavFileException.
- The component Reencoder uses LAME codec to encode and decode audio files. It is implemented by the Session Bean ReEncoderImpl.
- TagWatermarking applies a watermark on a mp3 file by adding a mp3 tag to it. It is implemented in the class TagWatermarkingImpl.
- When multiple files are at the same time downloaded, they are packaged by the Packaging component into a single Zip file. This is done by the Session Bean PackagingImpl.
- The Cache component is implemented in the stateless Session Bean CacheImpl, which makes use of the singleton Session Bean CacheSingleton.

3.3.4 Mediastore.basic

Mediastore.basic is a simple java project containing diverse utilities that are needed by all components. It is therefore always deployed along with the EJBs and packaged inside each EAR.

- The packages edu.kit.ipd.sdq.mediastore.basic and edu.kit.ipd.sdq.mediastore.basic.config contain classes implementing the EJB configuration. These are presented in more detail in Sections 3.4 and 3.5.
- The package edu.kit.ipd.sdq.mediastore.basic.data contains classes that represent data, which is usually used in the communication between different EJBs.
  - The class AudioFile represents the main structure of an audio file. It has two attributes of the types AudioFileInfo and FileContent.

  - The class AudioFileInfo holds a set of information about an audio file such as the artist, the genre or the ID of the original uploader.
  - The abstract class FileContent is an abstraction of the content of an audio file. It is extended by the two concrete classes FileContentLocal and FileContentRemote and contains two methods: the abstract boolean method isLocal, which tells whether the FileContent object is actually a FileContentLocal or FileContentRemote object, and the method convertIfNeeded, which converts the current FileContentLocal object to a FileContentRemote object or vice versa when a conversion is necessary.
  - The class UserRegData represents the user data upon registration.
  - The class CurrentUser represents the current user being logged in.

- The package edu.kit.ipd.sdq.mediastore.basic.exceptions contains exceptions that might be thrown by the application. The class AppException serves as a base class for other exceptions
- The package edu.kit.ipd.sdq.mediastore.basic.interfaces contains all business interfaces used in Media Store. IBusinessInterface is the base interface of all others. It is an empty interface that extends the interface Serializable from java.io. A detailed convention regarding the interfaces is presented in section 3.5.
- The package edu.kit.ipd.sdq.mediastore.basic.utils holds different utility classes that are used by almost all EJBs.

Other than the source files, mediastore.basic contains the configuration files ejbconfig.xml and GlobalConstantsContainer.properties at its root. It also contains the LAME codec as an executable file at the folder mediastore.basic/lame.

3.4 Reconfiguration

The main issue with Media Store 1.0 is that the connection between components is hard-coded. This means it cannot be changed without modifying the source code and redeploying the application. For example, suppose that a the watermarking component is deployed on multiple servers. To determine which server to use when watermarking audio files, it was necessary to specify the address of the server in the source code. To avoid that, the reconfiguration was rethought for more flexibility. In fact, it became possible to modify the components' connection settings at runtime without the need of redeploying.

The new reconfiguration system reads the XML file ejbconfig.xml[8], which can be found at the root of the project mediastore.basic and must be copied in the config folder of the running domain. This file contains all the information a component needs to communicate with other components. Components, in our case Enterprise Java Beans, are represented through the node <EJB name=""></EJB> where the attribute name is the name of the EJB in question. The EJB node comprises several other sub-nodes as listed below :

[8] The XML structure and its associated Java implementation were first developed in Media Store 1.1. They are described here in their final form as in Media Store 1.2

- <host> : IP address or hostname of the server on which the EJB is deployed
- <port> : This is the ORB Port associated to the server
- <appName> : JNDI application name
- <moduleName> : JNDI module name
- <beanName> : JNDI enterprise bean name
- <providedInterfaces> : List of the interfaces provided by this EJB
- <requiredInterfaces> : List of the interfaces required by this EJB

To illustrate this, let us take the example of the cache component (Listing 3). The cache EJB provides two interfaces IDownloadCache and ICacheMaintenance and requires the interface IDownload which is used to get the files to be cached. In this example, the files are provided directly by the component mediaaccess. To get watermarked files instead, one can simply replace mediaaccess with audiowatermarking and IDownloadMediaAccess with IDownloadAudioWatermarking, provided that audiowatermarking is already deployed. Another main advantage of this feature is to switch between servers on the fly. Suppose that cache is deployed on multiple servers at the same time. Simply changing host and port determines which server to use to cache files.

This XML structure is implemented in Java by the three classes EJB, ProvidedInterface, and RequiredInterface which are located at the package edu.kit.ipd.sdq.mediastore.basic.config, and is controlled by the class Config. In addition to that, the class BaseEJB located at edu.kit.ipd.sdq.mediastore.basic was implemented to serve as a base to all EJBs[9]. In order to communicate with one another, EJBs have to extend the class BaseEJB and call the method <T extends IBusinessInterface> T initRequiredInterface(String requiredInterface, Class<T> type). This method makes a call to the static method loadConfig() in the class Config, which checks whether any changes occurred on the configuration file by comparing its timestamp. If so, the file is deserialized from XML to Java objects. This deserialization is done by the XStream API available at the package com.thoughtworks.xstream.XStream. After deserialization, a JNDI lookup is made to the provided interface, which is assigned to the required interface. Further, the loaded interface resulting from the JNDI lookup is stored internally, so that it would be immediately available in the future, and thus avoiding making an unnecessary JNDI lookup in case the interface configuration has not changed.

Also worth noting is that the whole process of reconfiguration may eventually be turned off before deploying by setting the static variable reconfigurable to false. Doing so will make the system load the configuration only once, implying that it won't change during runtime.

To sum it up, the described reconfiguration system brings design and performance improvements in comparison to Media Store 1.0. As mentioned before, it brings the possibility to change the interface provided to an EJB on the fly. It also allows to switch the server an EJB uses, in case this one is deployed on two or more servers. Another main improvement is the reuse of an already loaded interface, thus lowering the number of JNDI lookups. At the same

[9] In the same way, the managed beans in mediastore.web use this system through the intermediary of the class WebBeanManager located at the package edu.kit.ipd.sdq.mediastore.web.beans.

```xml
<entry>
  <string>cache</string>
  <EJB name="cache">
    <host>localhost</host>
    <port>3700</port>
    <appName>mediastore.ear.cache</appName>
    <moduleName>mediastore.ejb.cache-1.0</moduleName>
    <beanName>CacheImpl</beanName>
    <providedInterfaces>
      <entry>
        <string>IDownloadCache</string>
        <ProvidedInterface>
          <providingEJBName>cache</providingEJBName>
          <name>IDownloadCache</name>
        </ProvidedInterface>
      </entry>
      <entry>
        <string>ICacheMaintenance</string>
        <ProvidedInterface>
          <providingEJBName>cache</providingEJBName>
          <name>ICacheMaintenance</name>
        </ProvidedInterface>
      </entry>
    </providedInterfaces>
    <requiredInterfaces>
      <entry>
        <string>IDownload</string>
        <RequiredInterface>
          <name>IDownload</name>
          <providedInterface>
            <providingEJBName>mediaaccess</providingEJBName>
            <name>IDownloadMediaAccess</name>
          </providedInterface>
        </RequiredInterface>
      </entry>
    </requiredInterfaces>
  </EJB>
</entry>
```

Listing 3: Example Deployment Configuration

time the reconfiguration comes with an obvious downside, which is reading and deserializing the XML file, or at least checking its timestamp.

3.5 Intelligent Remote and Local EJB Calls

Media Store 1.0 and 1.1 come with the inconvenience that all EJBs implement only remote interfaces. This adds a significant overhead whenever locally deployed EJBs call each other due to the parameter marshalling/un-marshalling. In addition to that, the content of an mp3 file has to be loaded in main memory each time it's passed as a parameter to an EJB, even if both EJBs are located locally, in which case it can be done far more efficiently.

Remote call overhead

When using only remote interfaces, the overhead caused by the parameter marshalling/un-marshalling will always occur, even if both the caller and callee EJBs are local to one another. That means if both are located on the same server and in the same EAR (**E**nterprise **AR**chive). To remedy that, a remote/local switch system was implemented. The system detects automatically whether the two EJBs are local, and if so uses the local business interface instead of the remote one. For this purpose, from each business interface in Media Store 1.0 the annotation @Remote was deleted, and a new remote and a local interface, that extend the original interface, were created. A JNDI lookup to the remote or local interface would then be made, based on the outcome of the boolean variable localCall:

```
1 boolean localCall = ejb.getAppName().equals(callee.getAppName()) && ejb.getHost().equals
     (callee.getHost());
```

For example, say the required interface is IFacade. By convention, the remote interface is IFacadeRemote and the local one is IFacadeLocal. If localCall holds true, the string "Local" would be appended to the required interface name, and so making a JNDI lookup to the local interface. Furthermore, the namespace java:app/ would be used instead of java:global.

Path rather than byte array

When two EJBs are remotely located to one another and an audio file has to be transfered, it is necessary to pass the file content in some methods like upload or download as a byte array. In the local case, this is no longer needed, since both EJBs are on the same server and can access the file in question directly on the hard drive. For this reason, a new abstract class FileContent was created in the package edu.kit.ipd.sdq.mediastore.basic.data. This class represents the content of a file in general. It has a method isLocal which returns true if the file is located on the hard drive, and false if the file content is an in-memory byte array. From this abstract class inherit the two classes FileContentRemote and FileContentLocal. FileContentRemote has the attribute bytes of the type byte[] and FileContentLocal has the attribute path of the type Path. Besides, the class FileContent has a method convertIfNeeded, returns a FileContentRemote or

FileContentLocal version of the FileContent based on a boolean flag passed as a parameter, that indicates whether the target file should be local or not. In addition to that, the attribute content of the type FileContent has replaced the byte array in the AudioFile class.

By doing so, this form of abstraction allows different EJBs to use the local version of FileContent to communicate locally with each other, or to switch to the remote version, if they are located remotely.

Quick Developer Guide

For developers who want to add a new EJB to Media Store, the following steps should be taken into consideration:

- Begin with creating the EJB.
- The implementation class must inherit from BaseEJB.
- Create a new interface in edu.kit.ipd.sdq.mediastore.basic.interfaces. This interface must extend IBusinessInterface. The methods provided by the EJB must be declared here.
- Create two other interfaces that follow this convention :
  - One must be annotated with @Remote, the other one with @Local.
  - Both interfaces must extend the original interface.
  - By convention, the name of the remote interface must be identical to the name of the original interface and appended with 'Remote'.
  - By convention, the name of the local interface must be identical to the name of the original interface and appended with 'Local'.
- The implementation class must implement both remote and local interfaces.
- Add an entry in ejbconfig.xml for the new EJB.
- Any calls to other EJBs must be made through the method initRequiredInterface(String requiredInterface, Class<T> type)

4 PCM Model

The current version of Media Store PCM model is built for the Palladio 3.5 stable release[1]. There is also a branch containing a model that is kept up-to-date with the current nightly release. The model reflects the performance of the implementation versions 1.0. The performance of the implementation did slightly increase with version 1.1. The model should be still applicable for the implementation version 1.1. Version 1.2 does not match the performance of the model, as it is much faster when components are deployed on the same application server and in the same EAR.

4.1 Description

The PCM model of Media Store represents an early design model. It is not a highly calibrated model to validate the PCM. Thus, the model has high error margins (up to 97 %). However, the tendencies for modeled and explored design alternatives are correct and a high error is to be expected for an early design model. More calibration is, however, always possible to improve accuracy. In the case of this PCM model, the high errors result from the really fast services (e.g. login). During modeling they were treated with less rigor, as they are not as performance relevant compared to other services. The download functionality is more important, as it causes the most resource demands. The model accuracy for download is much higher (error below 9 %).

The model was built, as it would be built in the an early design stage. The implementation is non-existent. However, third party components and prototypical code exists. It is especially worthwhile to prototype infrastructure functionality, which is used by all components (e.g., JNDI lookup and remote calls). Using a load driver and a monitoring tool, resource demands can be extracted from the existing code and fed into the model. Resource demands can also be transferred to code, which is expected to generate a similar load.

To model the concrete CPU resource demands, the response time of a single functionality was measured in isolation. As an approximation, we used the response times as resource demands. This is plausible, as the use of a RAM disc removes waiting times due to storage device access from most functionality (except for the initial retrieval from the DataStorage). The following functionality was measured:

- Overhead of remote communication and JNDI lookups, which is easy to prototype. This was measured once and inserted into every SEFF for components that correspond to EJBs. This demand is constant.

[1] https://sdqweb.ipd.kit.edu/wiki/PCM_3.5.0

- Query building (by the EJB/JPA) and execution (by the DBMS). This was measured once and then inserted for both database accessing components (MediaAccess and UserDBAdapter). This demand is constant.
- The resource demand of packaging audio files into zip files. We measured this functionality, as we consider it already available in the design phase, as it is provided by the Java API. We measured this functionality using multiple files with varying sizes and modeled the resource demand using linear regression. Thus, this demand is proportional to the overall size of requested files.
- We measured the demand of reencoding and audio watermarking. Both use the LAME codec, which is readily available. Further, we assume, that the audio watermarking component is a cots (component of the shelf). Without that assumption, we would have to estimate the performance of the component, which would lead to a worsened accuracy. An educated guess could be to double the demand from encoding, as lame is used two times for audio watermarking.

While we are satisfied with the accuracy of the model as an early design model, we are aware of much improvement potential. This could be addressed if one needs a highly fitted model for performance evaluation.

- The overhead due to transfer of audio data between EJBs is not considered. This is relevant for the upload and download services.
- All DB queries have the same constant resource demand. getFileList queries, however, should depend on the amount of entries in the database.
- The storage device (e.g., HDD) access is not modeled. This could be done for the DB and DataStorage Component. However, storage access is not easily approximated, as it does not work as a FIFO, nor as a processor sharing resource. Quite elaborate solutions are the use of performance curves [3], regression or queuing models [2].
- The resource demand of reencoding and audio watermarking is dependent on the requested audio files. However, a correlation which can be expressed in closed form is not easily found due to many influencing factors (input size, output size, lenght, bitrates, etc.). Thus, their resource demands are currently modeled using constant probability density functions, which are tuned to the audio payload and a uniform disribution of requests. The resource demands could be parametrized by using performance curves, regression or queuing models.
- The overhead of remote EJB communication and JNDI lookups was measured under a high parallel load. Later, this turned out to be problematic, as the expected workload utilizes all services and thus accelerating factors from the high parallel load of one service were no longer in effect (e.g., cache effects).
- With regard to the architecture, the model does exclude the web interface. The system is modeled from the facade component onwards which is used by the web application to dispatch requests into the backend. For calibration and validation, the performance was measured directly at the facade. However, except for effort, nothing speaks against the creation of a performance model of desired complexity and accuracy for the web part.

Resource demands are specified in nano load units, as they were measured in nanoseconds. The processing rate of the CPUs is defined as 1.000.000 nano load units per simulated time unit. Thus, the time unit for the simulation corresponds to one real world millisecond.

4.2 Audio Payload

The audio payload[2] under that the model was calibrated consists of 6 files, which were replicated 20 times each. The files are from the artist Trash80[3]. They are released under the Creative Commons BY-NC-ND 2.5[4]. The characteristics of the audio files are listed in Table 4.1. The replication was used to minimize caching effects that would have occurred if the size of the payload was to small. The amount of files per requests may be varied. However, due to the modeling of the reencoding and audio watermarking resource demands, the files have to be requested with equal probability. For our purpose, which is to have an early design time model, this modeling is sufficiently representative of a random MP3 collection. It was not our intent to create an intricate model of a MP3 collection. However, if desired, a model of arbitrary complexity may be created and the resource demands may be recalibrated or a more complex modeling of these resource demands can be put in place which is parametrized.

Filename	Title	Size (Bytes)	Bitrate	Length (Seconds)
a.mp3	Excuses	8.319.845	192	346
b.mp3	Pain Fade Down	5.192.682	192	216
c.mp3	Impact of Silence	5.316.819	192	221
d.mp3	Lazerscale 2010 - 01	4.901.973	192	204
e.mp3	Missing You	6.178.906	192	257
f.mp3	Within Time	11.290.941	320	280

Table 4.1: Audio Payload

4.3 Hardware Setup

The hardware setup, which is reflected by the model, is illustrated in Figure 4.1. It consists of two servers which are connected by gigabit ethernet. However, most of the architecture configurations only use one server. The database and file storage are located on the first server. The specs of the two servers are shown in Table 4.2. Everything is identical except for the cache architecture configuration for which the main memory of server one was doubled to make room for the in-memory cache. This did not accelerate the system substantially, as without the software cache the memory was no bottleneck in the 3 GB setup. The load driver is located on

[2] https://svnserver.informatik.kit.edu/i43/svn/code/CaseStudies/MediaStore3/trunk/Data/Audio%20Payload
[3] http://trash80.com/#/music
[4] https://creativecommons.org/licenses/by-nc-nd/2.5/

an additional machine. Latency between the load driver and the application server is irrelevant, as the measurements are taken directly on the application server.

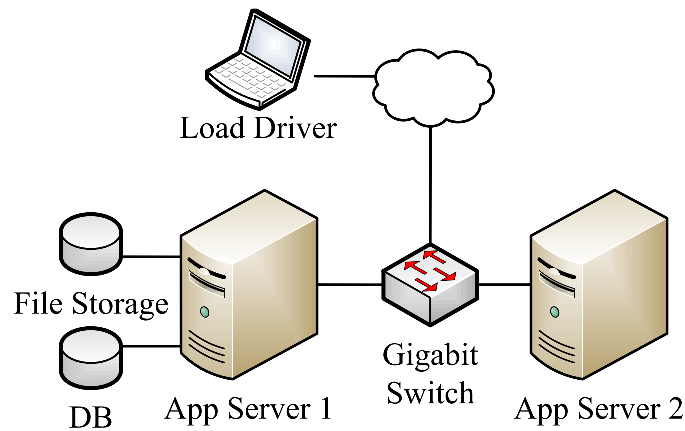

Figure 4.1: Deployment

Server	CPU	Memory	Bitrate
1	Intel DualCore E2180, 2GHz	3 GB / 6 GB	Windows 8.1 Update
2	Intel DualCore E2180, 2GHz	3 GB	Windows 8.1 Update

Table 4.2: Hardware Specification

4.4 Model Files

Figure 4.2 shows the current list of files of Media Store PCM model. In the Palladio book [1], four different configurations of the architecture are investigated. These configurations are:

base The basic configuration of the architecture. It features TagWatermarking and Reencoding. Every other configuration is a modification of base.

audio wm The audio watermarking configuration replaces TagWatermarking and Reencoding of the base configuration with the heavy weight AudioWatermarking. AudioWatermarking alters the audio signal of mp3s by inserting a inaudible watermark. This requires prior decryption and encryption after the watermark has been applied.

cache The cache configuration adds a software in-memory cache component between TagWatermarking and Reencoding. The cache stores a limited amount of reencoded audio files in memory.

distr reenc The distributed reencoding configuration sources the Reencoding component out to another server.

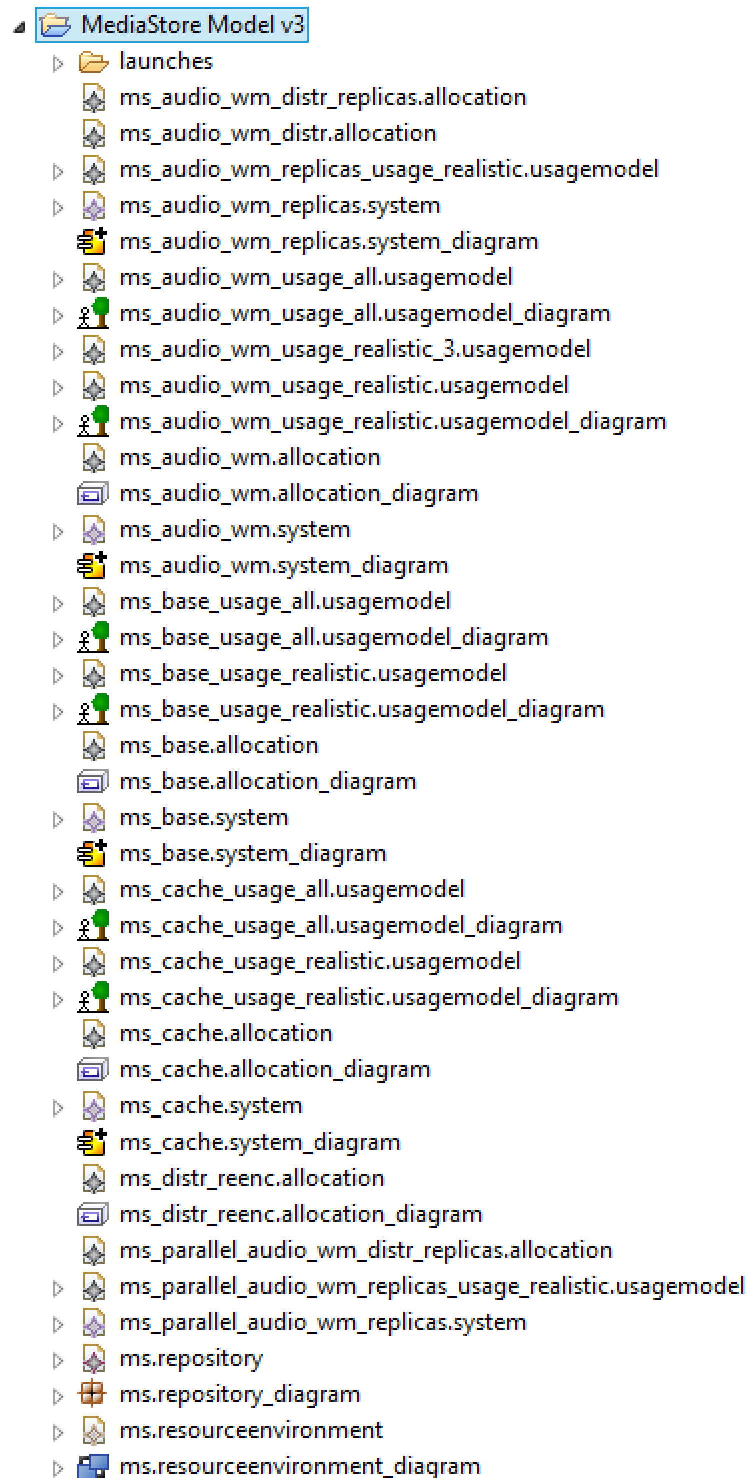

Figure 4.2: Files of Media Store PCM model

There is also a further configuration, parallel audio watermarking, which was not covered by the book. It runs the watermarking of files of one request on two CPU cores

The files belonging to the configurations are indicated by the corresponding prefix. The repository and resource environment are shared among all configurations. That is why they have no prefix. As the distributed reencoding configuration does only differ in its allocation from the base configuration, the two configurations share system and usage models.

There are two types of usage models provided for the configurations. The ones that are called "all" are quite simple. They feature a closed workload. A pass through the scenario behavior executes each service exactly once. These usage models were used for calibration and verification, as they induce no extra randomness. The other type of usage model is called "realistic". It was used to produce the results for the comparison of the design alternatives in the Palladio book [1]. It displays a more realistic flow of service calls with a probabilistic arrival time of the open workload, probabilistic branches and delays. To keep variance in check, the number of requested audio files to download is limited to one or two. The number is chosen randomly by a probabilistic branch. However, the model would even provide valid prediction results for an arbitrary number of files.

4.5 Launches

For each architecture configuration and each usage model, the launches have been persisted and provided within the modeling project. The are shown in Figure 4.3. They are named firstly after their architecture configuration, secondly after the usage model and lastly after the amount of measurements, which is to be taken. The simulation time stop condition is disabled in every launch. The launches which only take 1000 measurements should only be used for testing and not for retrieving prediction data.

4.6 Modeling of the Cache Component

The cache architecture configuration features a software cache component. The modeling of the hit rate is not straightforward, because it is dependent on the amount of files in the data storage, on the cache size and the way they are requested. There are two ways to model the hit rate, with regard to the distribution of which requested files. When an uniform distribution is assumed, it is possible to calculate the hit rate dependent on the cache size and amount of files in the data storage.

$$hit\ rate = \frac{cache\ size}{number\ of\ songs}$$

This does not apply until the cache has been filled. When doing performance measurements, this can be achieved by a warmup phase, which is not recorded. Other distributions may also be used. E.g., using a PMF (probability mass function) it is possible to categorize songs into categories of different popularity.

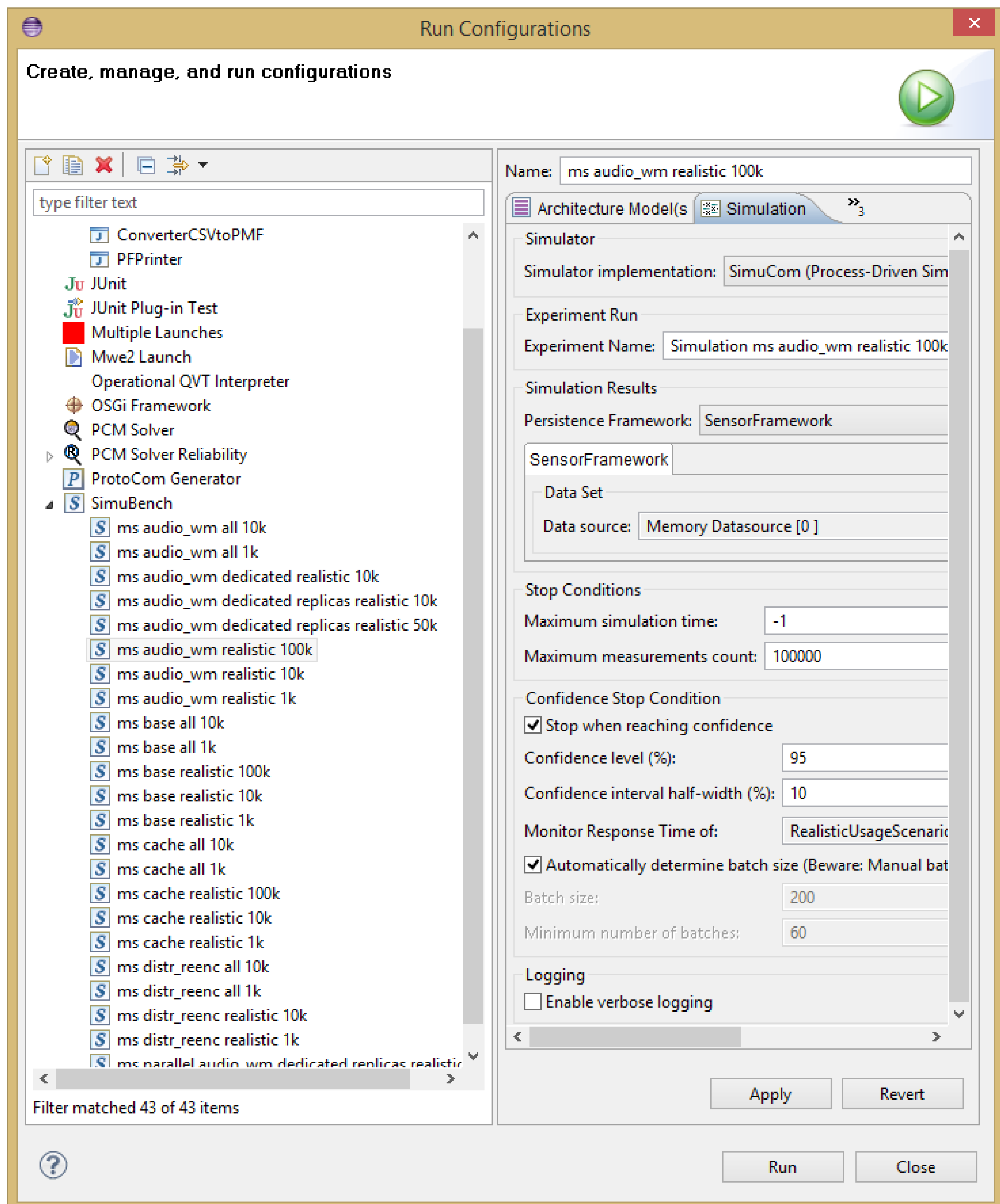

Figure 4.3: Run Configurations

The information needed for the determination of the hit rate should be modeled in the model for which the respective role is responsible. I.e., the amount of data and the request probabilities (or distribution) should be modeled in the usage model. This is, because the amount of files is also a result of the usage. Further, the size of the cache should be specified in the cache component.

For our experiment we chose to model and measure using the uniform distribution of requested files. Despite it not being the most realistic solution, we chose it because of pragmatic reasons. Modeling distributions of hit rates is easy. However, programming the load driver so that the hit rate is matched is not trivial. Inaccuracies caused by not achieving the hit rate will increase the prediction error, even though the problem is not caused by the model but the load driver. Decisive for our decision was the fact, that the purpose of the model is not an elaborate engineering of hit rates but early performance evaluation. One could, of course, model an arbitrary distribution. However, this will not be reflected in the resource demands, which are not parametrized with regards to file characteristics. These have to be re measured under the new distribution, or a more elaborate modeling has to be put in place, which takes file characteristics in account.

In our model, the hit rate is modeled by the content of the SEFF for the download operation of the Cache component. This is illustrated in Figure 4.4. In the PCM, probabilistic branches cannot be parametrized. I.e., the branch probabilities are static. Thus, the hit rate is not determined during simulation depending on cache size and amount of requested audio files. It is specified by a constant. Of course, this constant is calculated beforehand using the hit rate formula for uniform distribution.

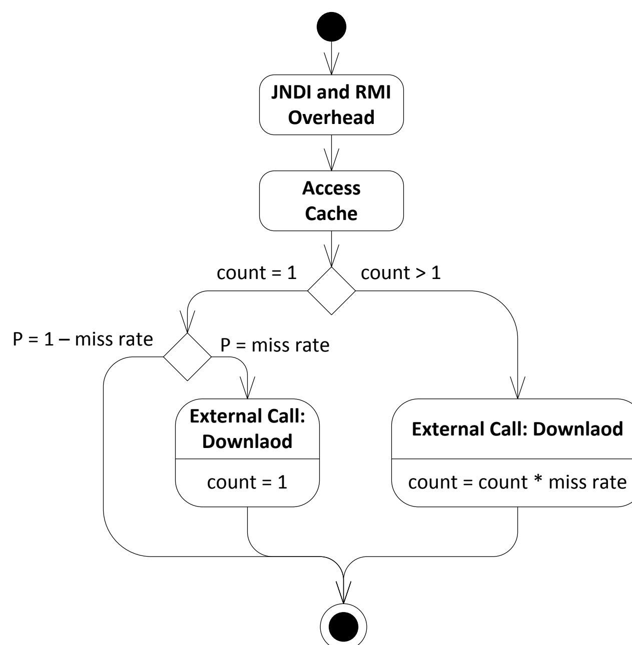

Figure 4.4: Cache: Download SEFF

In the case of multiple files being requested for download, the call should be kept as one and not split up into multiple calls. This can be achieved by multiplying the amount of requested audio files with the miss rate. Calls which request only one file for download should not be treated this way, as this would result in calls which request only a fragment of an audio file. This would not be realistic. That is why the SEFF first makes a distinction between the call requesting one or multiple audio files. In the case of one file being requested, a probabilistic branch is used.

5 Data

The data folder[1] contains: MP3 files of the audio payload, R scripts for error calculation, and calibration and evaluation data. Its content is shown in Figure 5.1.

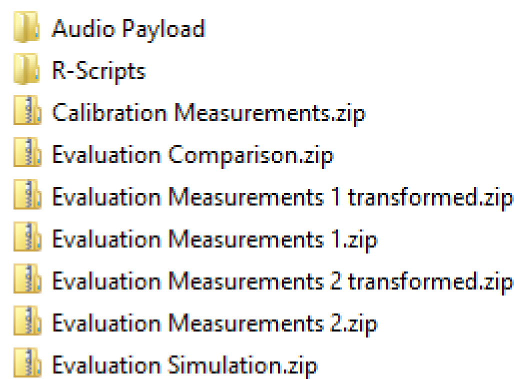

Figure 5.1: Data

The audio files used for the performance measurement, which are also modeled in the model's workload, are in the folder Audio Payload. They are from the artist trash80[2] and are licensed under Creative Commons BY-NC-ND 2.5[3]. Their characteristics are shown in Table 4.1.

Measurements from the implementation as well as prediction results of the simulation are stored in the form of data file sources. These can be opened using the experiments view in PCM stable 3.5[4]. It may still be possible to open the data sources with newer PCM versions. The simulation data is taken from Media Store model 1.0. The implementation data was measured from implementation 1.0.

There is one ZIP file containing a data source with calibration measurements. Table 5.1 shows the content. The first data source with calibration measurements was lost due to a data source corruption. However, the results are still reflected in resource demands of the model. Some rows are marked as invalid, that means that they were just tests, the measurement was disrupted, an exception occurred or the experiment was set up wrongly.

Table 5.2 shows content of the evaluation measurements from the implementation. There are two versions each. The original one contains measures in nanoseconds, the transformed

[1] https://svnserver.informatik.kit.edu/i43/svn/code/CaseStudies/MediaStore3/trunk/Data/

[2] http://trash80.com/#/music

[3] https://creativecommons.org/licenses/by-nc-nd/2.5/

[4] https://sdqweb.ipd.kit.edu/wiki/PCM_3.5.0

Index	Description
invalid	
2	TagWatermarking and Reencoding, uniform distribution of audio payload, single user, 100 minutes
invalid	
5	End-to-end measurement, full service coverage, 2 users, 2 hours
invalid	
10	getFileList, 30 min
invalid	
12	getFileList, 30 min, including Facade and MediaAccess
invalid	
14	getFileList, 30 min, including Facade, MediaManager and MediaAccess
invalid	

Table 5.1: Calibration Measurements

one contains measures in milliseconds for comparison with the simulation results, which are also in milliseconds. For all measurements, a load was applied, which matches the "all" usage model. It features a full service coverage an low degree of randomness.

Index	Description
Evaluation Measurements 1	
0	base, 1 user
1	base, 2 user
2	distributed reencoding, 1 user
3	distributed reencoding, 2 user
4	audio watermarking, 1 user
5	audio watermarking, 1 user
invalid	
Evaluation Measurements 2	
0	cache, 1 user
1	cache, 2 user

Table 5.2: Evaluation Measurements

Table 5.3 shows content of the simulation results. All measures are represented in milliseconds. Each simulation run terminated at 10.000 measurements taken. The "all" usage model was used for full service coverage and low randomness.

The evaluation comparison archive contains for each architecture configuration for one and two users: exported CSV files with measurements and the output of the error calculation script.

Index	Description
0	base, 1 user
1	base, 2 user
2	distributed reencoding, 1 user
3	distributed reencoding, 2 user
4	audio watermarking, 1 user
5	audio watermarking, 1 user
invalid	
8	cache, 1 user
9	cache, 2 user

Table 5.3: Evaluation Simulation Results

Bibliography

[1] Steffen Becker et al. *Modeling and Simulating Software Architectures - The Palladio Approach*. Ed. by Ralf H. Reussner et al. to appear. Cambridge, MA: MIT Press, 2016.

[2] Qais Noorshams. "Modeling and Prediction of I/O Performance in Virtualized Environments". PhD thesis. Karlsruhe Institute of Technology (KIT), 2015. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000046750.

[3] Alexander Wert, Jens Happe, and DennisWestermann. "Integrating software performance curves with the palladio component model". In: *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*. ACM. 2012, pp. 283–286. URL: http://dl.acm.org/citation.cfm?id=2188339.